Multi-Modal Routing and its application to MobilityDB

Master Thesis

Maazouz Mehdi

Master thesis submitted under the supervision of
Prof. Esteban Zimányi

Academic year
2021-2022

In order to be awarded the Master's programme in
Computer Science

Abstract

In view of the emergence of new transport modes and the climatic and political issues at stake, multi-modal routing is becoming increasingly popular and is therefore being studied. It brings many challenges compared to simple single-modal routing. The notion of the best route is becoming abstract and needs to be redefined. For this purpose, many algorithms are studied, a good part of them being derived from Dijkstra's algorithm and called Dijkstra's speed-up techniques. In addition, multi-modal requires a rethinking of networks designs and how they should be connected. In parallel, many data formats have emerged to group the different modalities. These are used to feed the various multi-modal tools such as *Google Maps* or *OpenTripPlanner*, an open-source competitor. This master thesis will therefore focus on presenting an overview of multi-modal routing and the tools that revolve around it. Besides it will provide *OpenTripPlanner* a more complete management of the *GBFS*, which focuses on shared mobility (shared bikes, shared cars, ...). Lastly, we will connect *OpenTripPlanner* to *MobilityDB*, an open-source moving object database.

Keywords: Multi-Modal Routing, OpenTripPlanner, GTFS, GBFS, NeTEx, Spatio-Temporal Database, MobilityDB

Acknowledgement

I would like to thank Esteban Zimanyi who has been an extraordinary supervisor with his constant availability. He was a great help in understanding the subject of multimodality on the one hand, and on the other, for guiding me throughout this thesis.

Contents

Chapter 1

Introduction

1.1 Introduction

Nowadays, we have several kind of transport, as a consequence the transport network has become more complex than ever. We could quote shared cars, shared bikes, e-scooters in addition to the classic public transport (tram, bus, metro).

For years, there have been some tools capable of doing computation in order to provide a path between two places (or more) to a user. These tools provided a path with cars, bikes, public transport or a mixed path with bikes and public transport as means of transport (see 2.1.5).

However, with the emergence of new modes of transport and changes in the way users travel, it becomes important to reflect on the principle of multi-modality..
Besides the current politics are tending towards a smart mobility to reduce the emission of gazes, to get cities less noisier and to reduce traffic.
Hence the multi-modal routing has gained more interest lately and nowadays the multi-modal routing is become an important field in computer sciences. Many tools have been developed around multi-modal routing with the aim of providing users with routes that include several modes of transport, all according to the users' preferences. Whereas some prefer a low number of transfers, others want to optimise the duration of the journey or want to get the cheapest journey.

This leads to consider of increasingly complex scenarios, leading to the design of specific pathfinding algorithms to return a suitable path to the user in time T, but also to data management techniques to reduce the amount of data to be processed upstream, thus reducing the execution time of pathfinding algorithms.

Chapter 2

Background and Definitions

2.1 Background and Definitions

Multi-modal routing has its roots in single-modal routing. The idea behind the single-modal routing is simple. It's the calculation of a path between two destinations using a single mode of transport.
In order to do this, the network is represented by graphs. As a plausible reminder we are going to introduce the needed underlying concepts of graph theory.

2.1.1 Graph Theory

A graph is represented by a tuple $G = (V,E)$ where V is a set of vertices and E the set of edges. Each edge is defined by a pair of vertices. An edge going from a vertex a to b is written as (a,b).
In the context of this thesis, we only use directed graphs which can be defined as graphs where the set of edges (E) is composed of *ordered pairs* of vertices. Moreover, we use the notion of cost which defines here the duration of the trip (it defines more than that when we'll discuss multi-modality). The cost is useful when we use algorithms to find a path.
The reverse graph $\overline{G} = (V,\overline{E})$ is the graph obtained from G by substituting each $(u,v) \in E$ by (v,u).
Among others notations, we have $|V|$ and $|E|$ which represent respectively the number of vertices and the number of edges.

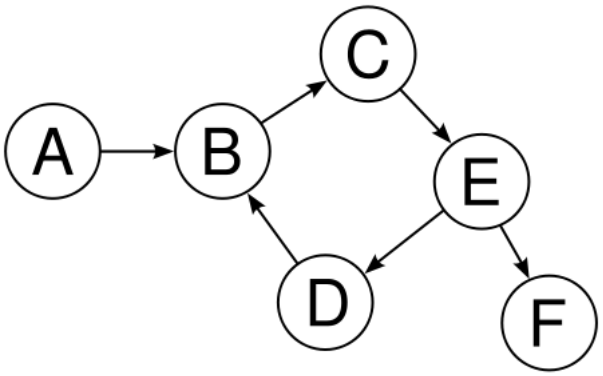


Figure 2.1: Picture representing a directed graph [1]

Some graphs have weights on their edges and are called *Weighted graph*
This problem has a well-known solution, namely *Dijkstra's algorithm* [16].

In general, graphs are widely used to model and study the structure of many networks (internet networks, social networks, ...). The vertices represent people, routers, ...

The edges represent the links between them.
These graphs have been widely studied over the years. However, there is a type of graph, called **temporal graphs**, where fundamental problems have not yet been well studied (reachability, shortest path, ...) [36].
An edge in a **temporal graph** can be defined as a tuple (u,v,t,d) where u and v represent start and destination node, whereas t represents the departure time and d represents the duration [36].
These graphs are interesting because they allow to model several networks where the notion of time plays a role. In particular a public transport network. As mentioned earlier, these graphs pose problems such as reachability. Indeed, in a public transport network, some lines can be interrupted. Therefore, a person wanting to reach a station on this line could miss the last tram or bus and be prevented from reaching this destination. This would not be a problem in a **non-temporal graph** [36].

2.1.2 Shortest Path Problem

The shortest path problem is the problem of finding a path between two nodes so that the cost of doing so is the minimum necessary. As a reminder we want to minimize the duration trip. This problem can be solved by Dijkstra's algorithm or one of its variant.

2.1.3 Dijkstra's algorithm

The Dijkstra's algorithm is an algorithm used in graph theory. Indeed, it allows to determine the shortest path between nodes designed by the computer scientist Edsger W. Dijkstra. We will explain how it works as it is fundamental in single and multi-modal routing. Most tools use this algorithm or one of its heuristics to determine a shortest path between nodes.

The algorithm starts from the starting node and will use a table that it will update during its execution. This table contains the shortesth distances between all the nodes to the starting node.

Initially, the distances are infinite except for the distance from the starting node to itself which is set to 0.

Then, we will update this table by adding to the adjacent vertices to the starting node the weight of the edges connecting them to the starting node. The neighbour visited by the algorithm will be the one whose distance is the minimum and which is unvisited, always with respect to our starting node.
Then the algorithm starts again at the visited node and will update the table by adding the distances from this node to its unvisited adjacent neighbours. If a distance already exists in the table, it will only be modified if the new one calculated is smaller. The visited node will be the one whose calculated distance is the minimum since the beginning of the execution of the algorithm.
By way of remark, at each step, the newest visited node will become the *previous* node of its adjacent nodes if and only if their distances are modified.
The algorithm continue its execution until all nodes have been visited. Dijkstra's algorithm is guarantee to be optimal and its execution in worst case complexity is $\mathcal{O}(|\mathrm{E}| + |\mathrm{V}|\log|\mathrm{V}|)$.

Let's go into an exemple where the node A will be our starting node and the node E will be the ending node.

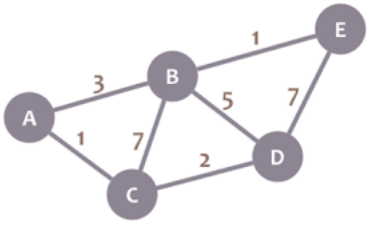


Figure 2.2: Graph representation [2]

The table before the execution of the algorithm.

Node	Distance	Previous
A	0	-
B	∞	-
C	∞	-
D	∞	-
E	∞	-

As explained, the algorithm starts by visiting A and updates the table by adding the distances of adjacent neighbours (the line A is not necesserary, so we can remove it).

Node	Distance	Previous
B	3	A
C	1	A
D	∞	-
E	∞	-

The newest visited node is C. Regarding the table, we will update it by respect to C.

Node	Distance	Previous
B	3	A
C	1	A
D	3	C
E	∞	-

B is not modified because its distance from A is smaller. Here we have two choices because the distances from A to B and from A to D are the same. Let's suppose we force our algorithm to visit in alphabetical order. So we continue by visiting B.

Node	Distance	Previous
B	3	A
C	1	A
D	3	C
E	4	D

Then we arrive at E our ending node but there is still unvisited node. We have to take it into consideration before finishing our algorithm. Here the unvisited node is D and it's an adjacent node to E. So we visit it.

Node	Distance	Previous
B	3	A
C	1	A
D	3	C
E	4	B

All nodes are visited and the shortest distances from them to the starting node are calculated. Now, in order to group the nodes that are part of the shortest path between the ending and starting nodes, it is sufficient to go up the nodes by the previous column.

2.1.4 Speed-up Techniques

When you want to do multi-modal routing or intercontinental trip, the Dijkstra algorithm becomes too time consuming. In this case, alternatives must be found to reduce the time needed to obtain a route [13].
The speed-up techniques of the Dijkstra algorithm enable to reduce the number of nodes visited and thus, to reduce the necessary running time. However, this is at the expense of the optimality of the solution.
In order to be faster, speed-up techniques usually use preprocessing and some of them usually use heuristic.

We are going to describe 2 speed-up techniques well-known from the literature. The first one is A* which is a speed-up technique using heuristics to reduce running time.
The second one is the bidirectional search. Its particularity is to perform two searches, one starting from the ending node.

This is a non-exhaustive list, not only there are other speed-up techniques, but you can also combine several speed-up techniques together [30].

A*

The A* algorithm [23] is an extension of Dijkstra's algorithm that prioritises speed of execution over optimality of the solution. It adds a cost from a heuristic to the nodes.
Its execution is similar to Dijkstra's algorithm except that the next visited node no longer depends solely on its distance from the starting node but on the sum of this distance and the heuristic cost.
There are different heuristics that can be applied to A* like the Euclidean distance or the Manhattan distance.
If the heuristic does not overestimate the cost, the algorithm is considered admissible, in other words, it guarantees to find the shortest path between two nodes.

Bidirectional Search

An another speed-up technique based on Dijkstra is the Bidirectional Search [32]. Indeed it is going to perform Dijkstra's algorithm from the starting node to the ending node as well as it's going to perfom the Dijkstra's algorithm from the ending node to the starting node at the same time (we keep the same set V but we take the set of reverse edges $\bar{E} =$ (v,u) | (u,v) $\in$ E) .

The algorithm will stop as soon as we have a node y that has been visited by both Dijkstra's algorithms and the path going from the starting node to y is the sorthest path while the path going from the ending node to y is the shortest path as well.

Bidirectional Search allows to cut down the number of edges we have evaluated and as a result of reducing the running time.

2.1.5 Tools

Many tools, open-source or not, have emerged over the years. We review some open-source tools for single-mode routing. We will focus on one of them, namely *PgRouting*.

The main reason for this choice is that *PgRouting* is easily compatible with *MobilityDB* which is an extension to the PostgreSQL database system and its spatial extension Postgis. It allows to represent temporal and spatio-temporal objects and to store them in a database [38]. In addition, *PgRouting* can be easily extended to a multi-modal trip test.
Previous work highlighted this compatibility between *PgRouting* and *MobilityDB*, in particular by allowing the generation of trajectories corresponding to users moving from home to work [37].

Open Source Machine Routing

Open Source Mahine Routing[3] is a routing engine written in C++ that has several features. It's designed to run on *OpenStreetMap* data. It could find the fastest route between coordinates.

It takes into account 3 means of transport, namely car, bicycle and walking. OSRM offers a graphical interface as a demo to test its capabilities. It is a project that continues to evolve (its last update was in May 2021). Unlike *PgRouting*, there is no need to install several tools before.

Once the capabilities have been tested via the demo, the use of OSRM is done via its API. Some interesting options such as the exclusion of certain routes (motorways) or the desire to find alternative routes for a journey can be included in the request.

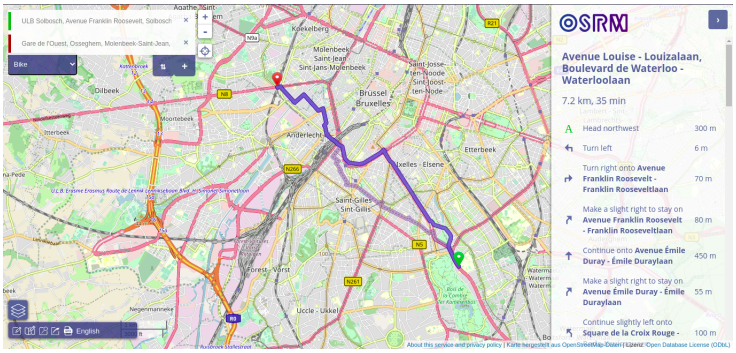


Figure 2.3: graphical interface of OSMR demo

OSRM offers two types of algorithms for route calculation, namely *Multi-level Dijkstra* and *Contraction Hierarchies* [20].

Regarding *Multi-level Dijkstra*, it corresponds to the implementation of a part of a overall method, called *Customizable route planning* [13] but differs by its partition phase [4].

[3] https://github.com/Project-OSRM/osrm-backend

[4] The partition steps are implemented by InertialFlow instead of PUNCH because of patent issue: https://github.com/Project-OSRM/osrm-backend/issues/4797

GraphHopper

GraphHopper [5] is an another routing engine written in java which includes several means of transport as cars, bike, foot, scooter, and public transit (as well as small trucks, racing bikes) By default, it uses data from *OpenStreetMap* as well as *GTFS* data (for public transit) although it could import data from other sources.
Graphhopper can be installed locally or can be used by a web interface. As well as used as a service from other languages like Python, Ruby and so on.

Graphhopper uses several algorithms to calculate a path, namely Dijkstra but also speed-up techniques such as A* (and its bidirectional variants), CH [20], LandMarks [21] when the path computed is too big (intercontinental for example).
There are several modes to calculate a trip. The first is called "speed mode" and uses the CH algorithm and has the advantage of being fast and not using heuristics. Unfortunately, the pre-processing step is time consuming and resource-intensive.
The second one is called "hybrid mode" and uses the Landmarks algorithm. This mode also requires preparation time and memory, but it is much more flexible regarding changing properties per request [6]. Beside it uses less RAM than the "speed mode".

An interesting remark is that Landmarks is currently limited, indeed it is not possible to cross the borders between the EU, Africa, and Asia. Indeed this limitation is important as we store the weight approximation used as heuristic by Landmarks in a short (two bytes) and for large distances more bytes would be necessary. As an example for the world wide case this would mean several additional GB per weighting. [7]

Regarding the properties, you can avoid some roads (primary roads, motorway, ...) but also more specific properties. For example, avoid cobblestones if you are in a wheelchair or exclude the bridges.

PgRouting

PgRouting is an open source library which *extends the PostGIS / PostgreSQL geospatial database to provide geospatial routing functionality*[8]. It provides routing features like Dijkstra's algorithm, A*, K-shortest path algorithm that allows to compute path between points.

PgRouting requires many tools to be functional. In our case, we used a PostgreSQL database with its PostGIS spatial extension installed and pgrouting on top. The mapping of Brussels is provided by *OpenStreetMap*. In order to extract the data from *OpenStreetMap* and import it to PostgreSQL/Postgis we use the command line tool *osm2pgrouting* and *osm2pgsql*.

Thanks to these tools, we have created relationships (table *ways*) in our database in which we can identify interesting attributes that will allow us to do single-modal routing [22]

- the coordinates of the nodes
- id of the nodes
- coordinates of source and target nodes connecting the edge
- length of edges

[5] https://www.graphhopper.com/
[6] https://github.com/graphhopper/graphhopper
[7] https://github.com/graphhopper/graphhopper/blob/4.x/docs/core/landmarks.md
[8] https://pgrouting.org/

- maximum speed of edges
- if an edge one way (if there exists a wrong way especially for car network)
- the cost on an edge (and its reverse cost)
- the type of the edge (motorway, living street, ...)
- the geometry type which is important to drawn the node, the edge, the polygon,... on the map

The results of the *PgRouting* queries are displayed by a *QGis* client which is *afull-featured, user-friendly, free-and-open-source (FOSS) geographical information system (GIS) that runs on Unix platforms, Windows, and MacOS.*[9]

Here is an example of a journey from a node A to a node B in Brussels, computed by *PgRouting* (Dijkstra's algorithm) and using the road network.

Figure 2.4: Trip using road network and Dijkstra's algorithm

This example is made possible by the functions provided by *PgRouting*. Here we used **pgr_dijkstra** function. [10]. The way we get the road network is discussed below.

PgRouting also has a function simulating the A* algorithm which is much the same as Dijkstra's except that the view also requires the coordinates of our start and end vertices. In addition, we can specify the heuristic used.[11]
PgRouting has many more functions like **pgr_KSP** (K-Sorthest path), **pgr_bdAstar** (Bi-directional A* Shorthest path) or **pgr_johnson** (Johnson algorithm) and so on.

Remarks on tools

Of course, there are other tools for single-modal routing such as Valhalla [12], OpenRouteService [13], OpenTripPlanner (we discuss more about it in 5.1.1) and so on.
However, most of these tools are not adapted to multi-modal routing which poses several issues. We will see what issues multi-modal routing poses as well as what as the theoretical and

[9] https://github.com/qgis/QGIS
[10] https://docs.pgrouting.org/latest/en/pgr_dijkstra.html
[11] A* does not change the shortest path in 2.4
[12] https://github.com/valhalla/valhalla
[13] https://openrouteservice.org/

practical solutions provided by tools.

2.1.6 Networks

In order to apply algorithms such as Dijkstra, or its derivatives, we need to represent the different networks as graphs. That's the reason why we introduce an approach to model pedestrian, bicycle and road network in a single-modal routing [25].
The networks are built separately. Inside the pedestrian network, junctions are represented by nodes and if there exists a footpath between two nodes, then this footpath is represented by an arc. Arcs can also represent agriculture path, residential path and so on...

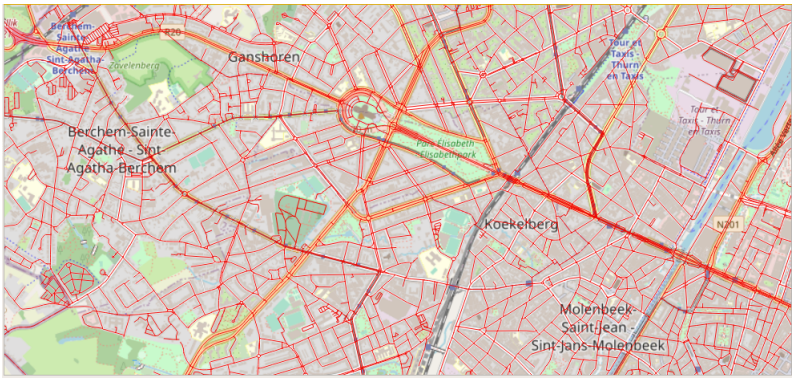
Bicycle network is similar to the pedestrian network. Indeed junctions are also represented by nodes and an arc is implemented whenever a bike is allowed to take the path. Of course there exists several kinds of path, for example a cycle path close to the road, a path which is only allowed to pedestrian and bikes, ...

Finally the road network is composed of nodes which represent junctions and edges which represent road. Here the notion of directed graph is important because of some roads are only one-way.
The cost could be the duration trip or the distance. In the case of a one-way road, the cost could be infinite in the wrong way.

2.2 Personal experiments with PgRouting

This section is about generating trajectories using *PgRouting* and connect *MobilityDB* on those trajectories. The visualisation is done by *Qgis*. The generated trips are multi-modal, more concretely, the trips use the street network , road network and we have thought about adding a transit network build by *GTFS* data[14].

Then we use the tool *PgRouting* and the map from *OpenStreetMap* to construct different newtorks and attempt an multi-modal network (this experiment is based on an similar experiment [22]).
To do so, we have to take several parameters into consideration, for example a pedestrian can use a path that is closed to cars, a one-way street or a specific pedestrian path. Conversely, a pedestrian cannot take the motorway, and in our case, the tunnels through the city of Brussels. The idea is the same if we decide to incorporate bicycle routes.



[14]Definition of GTFS format here: https://gtfs.org/

Figure 2.5: The road network of the city of Brussels drawn in red

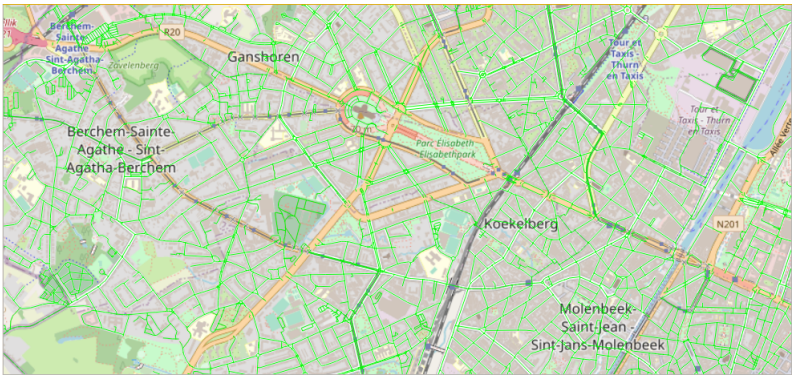


Figure 2.6: The pedestrian network of the city of Brussels drawn in green

Besides, our shortest path algorithms must therefore adapt to different networks. Our algorithms calculate a trip by trying to minimise the distance travelled or the time needed to get there.
The means of transport being different, the speeds of these different means are also different (a pedestrian does not go at the same speed as a bicycle or a car).

In our example below, our Dijkstra algorithm will use the time needed to get from point A to point B to calculate the cost of each arc. Each network will therefore have its own costs.

In order to calculate the costs for the pedestrian and bicycle network, we used the average speeds of 4km/h for a pedestrian (1.3m/s) and 15km/h for a cyclist (4.1m/s).
The cost of each arc could then be calculated as the division of the length of the arc in meters (obtained via the `length_m` attribute) and the average speed, also in meters [22].

The cost of the road network could be calculated by dividing the maximum speed allowed in each street by the distance of the street [15].

Furthermore, thanks to the tag `highway` parameter[16], we can know the type of our road (`motorway`, `primary way`, `residential road`, `...`) but this is not enough to construct our road, pedestrian and bicycle networks. Indeed, there are other parameters such as `foot`, `bicycle` to take into account. For example, `primary way` is used mainly by cars, however `bicycle` parameter can take several values such as `no` (no bicycle allowed), `yes`, `use side_path/designated` (road which has a compulsory cycleway).

Finally we applied the Dijkstra's algorithm by setting its directed parameter to true when calculating the route by road network. This made it possible to take into account one-way streets in the city.

[15] The street being equivalent to an arc
[16] https://wiki.openstreetmap.org/wiki/OSM_tags_for_routing/Access_restrictions#Belgium

Figure 2.7: Trip drawn by Dijkstra's algorithm: by car in red, by foot in green

We have used some of the code provided by the benchmark of BerlinMod [37] to convert our trip to *MobilityDB*. Thus, you can see a person (represented by a red dot) moving over time on a route calculated by *PgRouting*.

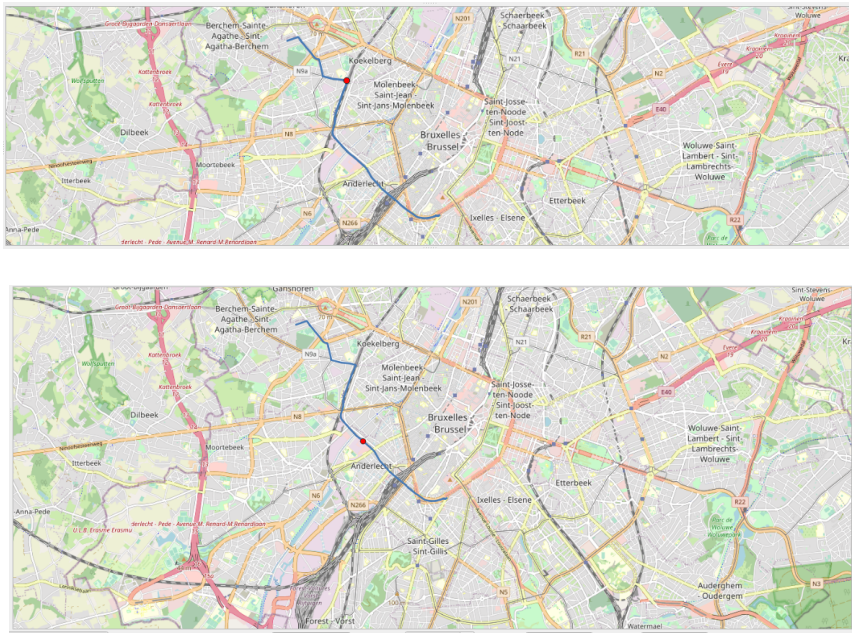


Figure 2.8: *MobilityDB point* on a trip computed by *PgRouting*

2.2.1 Multi-modal routing issues

As we have just seen, the calculation of a simple trip (i.e. with a single means of transport) is relatively easy. Once the network is obtained, we just need to apply a shortest path algorithm to see our route take shape.

We will now make some small changes to our configuration in order to practice multi-modal routing.
To do this, we added a type `attribute` containing the values `car`, `bike` or `pedestrian` to the different networks and took their union.
An example of a trip calculated by the networks union is shown in Figure 2.9, which shows the trip made, first by car, then on foot and finishing by car.

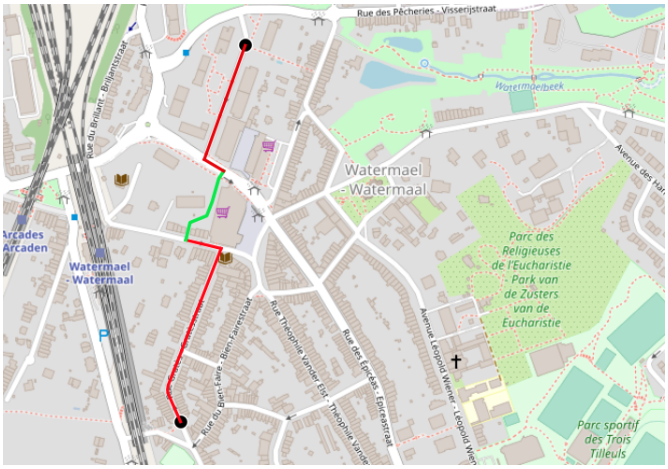


Figure 2.9: Trip with two modes of transport, part by car (red) and part on foot (green)

As can be seen, by naively merging the different networks, we end up with unwanted behaviours. It is not conceivable for a user to make part of his journey by car and then pick it up again to complete his journey.

2.2.2 Transit network into PgRouting

An attempt was made with *GTFS* data from the *Stib* agency to manage a transit network as well. We imported this data into PostgreSQL using an open-source tool *gtfs-sql-importer*[17]. Thus, we had the necessary topology to build a transit network into *PgRouting*.

However, after some reflection, many problems arose for us:

- how to connect our networks to the transit network?
- How to calculate the best public transport trip? Should we take the fastest one, the one with the least transfer, or a mix?

Moreover, *PgRouting* does not currently offer time-dependent algorithms[18]. Therefore, the Dijkstra algorithm (or any other provided by *PgRouting*) does not take into account the desired departure time, arrival time and all times provided by *GTFS* data. This lack can cause biased trips.

2.2.3 Conclusion on PgRouting

As a result, *PgRouting* is an efficient tool for single-mode routing. Moreover, it allows a certain freedom in the creation of different networks without having to deeply change the structure of the source code as it could be the case in OSMR or GraphHopper.

However, it is not suitable for multi-modal routing. Indeed, it is not enough to merge the different networks together, as this leads to unsuitable behaviour as seen above.
In addition, the establishment of a transit network poses problems, as mentioned above.

We are now focus on multi-modal routing and try to understand the solutions and tools provided to solve these kinds of problems.

[17] https://github.com/kausaltech/gtfs-sql-importer

[18] Actually, there exists time-dependents algorithms developed by students in the framework of the Google summer of code, designed for the first versions of *PgRouting*. The code dates back to 2011 and is now difficult to compile and not usable with a recent version of *PgRouting*: https://github.com/pgRouting/pgrouting/issues/448

Chapter 3

Multi-Modal Routing

3.1 Multi-Modal Routing

3.1.1 Observation on Multi-Modal Routing

As mentioned earlier, Dijkstra does not perform well on a huge dataset. Thus, many speed-up techniques have been developed to accelerate the computation of the path at the expense of a pre-processing phase. However some of those techniques suffer from a drawback, they only work for road or railway networks [14].

Unfortunately, one cannot simply merge the different networks as this leads to unintended behaviour. For example, a trip by car, then by foot and finally continue with the car. These issues make it necessary to review the structure of the network in order to find solutions.

3.1.2 Construct The Network

A solution, proposed by *Dominik Kirchler* [25] can be given by special edges. Indeed we will keep the different networks and connect them by those edges, called "transfer" edges, to which we will add a cost. This allows us to represent the cost of a change of transport mode (discomfort, money,...).
Pay attention networks will not share nodes, all nodes are duplicated and we will move from a network to another by these "transfer" edges. Besides, we will only connect each network to the foot network. As a consequence, there is no edges going from the bike network to the car network. We think this is rational because in real world, most transfers involve walking.

A distinction must be made between "transfer" edges. A vehicle should only be accessible at the place where it is located. This is why we can put labels on these edges (embark, disembark). The embark edges will connect the foot network to the others at the locations where a vehicle is located while the disembark edges will connect the networks to the foot network as far as possible (see below)
The difficulty is to know where to place these different edges, which nodes should connect them to avoid uncomfortable situations.

- **Access to Personal Car/Free Floating Car:** Theoretically, a car can be reached anywhere parking is allowed. This corresponds to a good part of the network if we remove motorways, tunnels and some fast roads. The principle is the same for free floating cars that can be dropped anywhere (by respecting the Highway Code). As a result we have to identify nodes from roads which allow the car park and connect them to their nearest neighbour node from the foot network. In Brussels some companies offer this kind of

service like *Poppy* [1]

- **Access to Shared Car:** The shared cars have the particularity of being located at specific places, the stations. In addition, they must be dropped off either at the same station or at another station, unlike free floating cars. As a consequence it's relatively easy to place the "transfer" edges. They will connect nodes from foot networks which are the nearest neighbors to nodes from car network (corresponding to these stations). In Brussels, some companies offer this kind of service like *Cambio* where each vehicle must be dropped off at the same station from which it came.

- **Access to Personal Bike/Free Floating Bike:** Bicycles can also be picked up/-dropped off in many places. Indeed, there are bicycle parking facilities, but nothing prevents the user from securing his bicycle with a lock in a street. As a result we have to connect every node (in road network and foot network) to its nearest neighbour in the bike network by a transfer edge. The principle is the same for free floating bike like *Jump* from *Lime* or *Billy Bike* in Brussels.

- **Access to Shared Bike:** Shared bikes are similar to shared cars except that they are part of the bike network. At each rental station, we will connect a node from the foot network to its nearest neighbour from the bike network (corresponding to the station). Again in Brussels, there is a company that offers this kind of service, it's called *Villo*.

- **Access to Public Transportation:** The networks will be connected to the public transport network at bus, tram and metro stations. As a cost, we could keep the money to get in public transport or the discomfort to change the means of transport but we could add the time needed to wait the bus or the tram for example.

Please note that there may be some inaccuracies. Indeed, in our case the nodes are located at road junctions. Moreover, the rental stations are associated with the nodes located at the closest road junctions, which could move them a few metres from their real locations. A more accurate solution would be to introduce additional nodes at the exact locations of these rental stations although the graph obtained would be bigger.

However this solution allows the incorporation of user preferences. Many criteria must be taken into account in real world applications. Indeed, the user may wish to avoid taking a means of transport, favour means of transport that emit little CO2, take low slope roads by bicycle and so on. All this is possible because no pre-computation is necessary, the calculation is done during the search for the shortest path with the weights on the egdes corresponding to the criteria favoured by the user.

[1] https://poppy.be/en

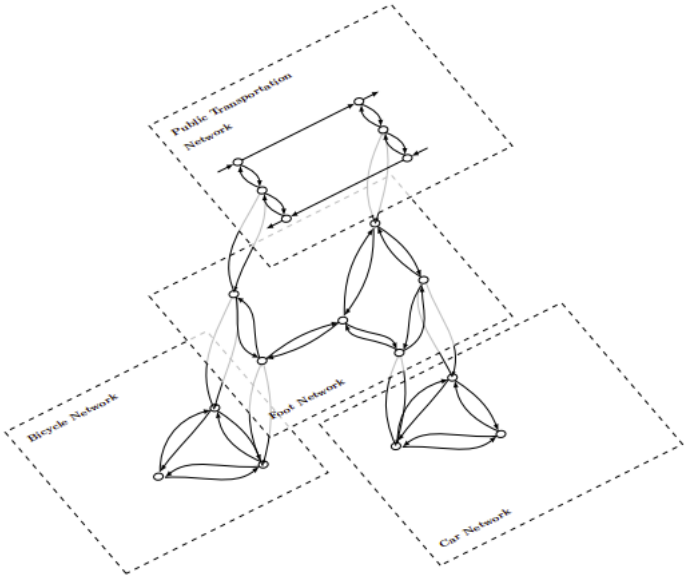


Figure 3.1: Overview of networks connected by these transfer edges [25]

3.2 Pareto Principle

The multi-modality implies a rethinking of the structure of our network. However, it also requires changes to the shortest path algorithms. Indeed, the fastest route is not necessarily the one that will be preferred by the user when travelling by train or bicycle. A longer train trip may be acceptable to the user if the user transfers less, or pays less for the trip. An another example is by bike, the user may prefer a longer but less inclined trip [11].

Therefore, multi-modal involves thinking about several criteria, hence the need to introduce the pareto-optimal path. Pareto optmiality could be defined as a measure of efficiency in multiobjective optimization [9]. This concept is applied on several applications like game theory, engineering, economics and so on.

3.2.1 Definition and Notation

To get more familiar with the concept of Pareto optmiality, you could find the definiton below:

- *A point x' $\in$ X with $f_i(x')$ is called Pareto optimal (or efficient or non-dominated, or non-inferior), if and only if there exists no point $x \in X$ such that $f_i(x) \leq f_i(x')$ for all i = 1, 2,...,k and $f_j(x) < f_j(x')$ for at least one index j $\in$ (1, 2,...,k).* [9]

- if we transpose our definition to the problem of multi-modal routing, a path is Pareto optimal if and only if there exists no (other) path performing at least as good as our first path for all criteria AND performing strictly better at least for one criterion.

- For example, given the Pareto-optimal set (9,3),(6,5),(1,8), then (5,4) would make it into the set because $5 < 9$ (comparing to element 1, criterion 1) and $4 < 5 < 8$ (comparing to element 2 and 3, criterion 2). However (7,6) is not Pareto-optimal because (7,6) is greater than (6,5).

A criterion can be represented as *time needed, price, steepness, number of transfers,....* The Pareto set (or Pareto frontier) is the set of all Pareto-optimal solutions.

3.2.2 Public Transit Network Routing

We also want to insist on the algorithms used in the public transport network. Previously we mentioned Dijkstra and his speedup techniques that speeding up queries through preprocessing phase (see 2.1.3 and 2.1.4). These speedup techniques can be applied to road networks. Unfortunately, they are not adapted when applied to public transportation network [4].

Indeed, unlike road networks, the best paths are based on several criteria and not only on travel time (hence our introduction of the pareto optimal path). In fact, several attempts with augmented versions of Dijkstra have been made [8], [11], [17], [28], [19]. However, the running time increased significantly and delays as well as cancellations made the preprocessing phases unsuitable.

Faced with these problems, preprocessing-based methods has been developed but making them efficient remains a challenge [6].
If we focus on the transit network, it turns out that there are less topological changes than metric changes. Therefore, algorithms have been developed that use the topology as a preprocessing phase.

Algorithms based on Raptor

Raptor belongs to a class of algorithms which are not Dijkstra based. It was introduced in 2012 [15]. It works in rounds and can be improved by prunning rules and multi-processing. Moreover it does not use a pre-processing phase.

In order to provide data to the algorithm, the user needs to acquire data from a public transit agency. These are expressed in different formats: *GTFS*, *GTFS-Realtime*, *NeTEx*,...(see 4.1)
The algorithm described in the paper is designed to give the Pareto-optimal paths by minimising 2 constraints. Namely the arrival time and the number of transfers.

Raptor algorithm can be extended to handle special cases. We have *Range Raptor* to handle bicriteria range queries. It's suitable when you want to go from a position to another between a time range. For example, if you want to travel from your home to your work between 07:00 and 08:00. Then *Range Raptor* will work in iterations over minutes, as a result it will start at 08:00, and for every minute until 07:00 (included), it will run a new search. The output will be the set of Pareto-optimal paths in such a period with respect to criteria (travel time and number of transfers).
Range Raptor can be make parallel as well. Note that if the set of departure time of trips at the source stop is greater than the number of CPU cores, it will be necessary to go through a partioning stage [15].

McRaptor (More criteria Raptor) allows *Raptor* to run with more than two criteria. In order to do this, each stop will be associated to multiple nondominating labels and the original algorithm is modified.

HypRaptor (Hyper Raptor) is a partition-based speedup technique for *Raptor* recently developed [12]. The idea is to partition the transit network by using suitable techniques for finding partitions. Then using *HypRaptor (Hyper Raptor)* which is a variant of *Raptor* to exploit those partitions and return paths. This version allows for less routes scanning and therefore offers better performances. However further researches have to be made in order to extend this algorithm beyond bi-criteria optimization.

Dijkstra Based Techniques

There are also algorithms based on Dijkstra suited for public transit network like *Layered Dijkstra*, a variant of *Dijkstra'algorithm* which is efficient when the optimisation criteria are the earliest arrival time and the number of transfers [6].

Transfer Patterns is a speedup technique [5] based on a preprocessing phase. Indeed many shortest paths have the same sequence of stations where the user has to change of vehicle (called transfer pattern). As a result transfer patterns are computed in a preprocessing phase. During the query phase, graph is represented by the union of transfer patterns between the source and target stops. The graph obtained can be processed by a *Dijkstra's algorithm*.
Transfer Patterns has an heavy preprocessing phase but query is processed fast. There exists an partition-based version of *Transfer Patterns* that reduces the preprocessing time [7].

3.3 Reachability Problem

Reachability problems have been a lot studied in graph processing over recent years, matching the advent of graph structures consisting of hundreds of millions of nodes and billions of edges. As a result, solving the reachability problem can quickly become a challenge [31].

Accordingly, some of the researches focus on partitioning and data sampling, two fundamental strategies used to speed up the computation of big data and increase scalability [26].

To be used on a large scale, the tools must be able to calculate reachabilities queries efficiently. Achieving this goal in public transport networks can quickly become a challenge because the shortest path between two nodes depends on the time of departure but also on the time needed to cover a path, which can vary considerably. Computing a index for public transport networks is more complex than pedestrian networks or road networks because edges cannot be traversed at any time and edge-traversal costs aren't constant [33].

Several indexing techniques have emerged for processing with temporal graphs [31], [35], [36]. However, index structures are heavy and those techniques do not scale [33]. Recent researches focus on partitioning directly the network into several regions/cells in order to use a index on them [33].

Chapter 4

Formats

4.1 Fomat Mobility

So that the various multi-modal tools can calculate and return a trip to the user, they need data on the one hand, but more importantly they need these data to be provided in a certain compatible format, a standard format being preferable.

There are a multitude of formats that have been created and used over the years. Some formats allow the representation of information from public transit agencies, while others focus more on shared mobility. Other formats, such as *NeTEx*, aim to bring together the two.
Below you will find an overview of the most popular, important formats for representing data needed for multi-modality.

4.1.1 GTFS

In the early 2000s, popular online mapping services such as *Google Maps* or *MapQuest* did not offer routes that included public transport. Moving around in unfamiliar cities could become frustrating. The car, supported by online mapping services, was therefore preferred [27].

GTFS (General Transit Feed Specification) was created in 2005 by Google and TriMet. It's a data specification that allows public transit agencies to publish their transit data in a compatible format with a lot of software applications [1].

It used by a variety of third-party software applications for many different purposes. An overview of these features including trip planning, timetable creation, mobile application, data visualization, accessibility, analysis tools for planning, and real-time transit information systems was proposed by A. Antrim, S. Barbeau [2].
The adotpion of *GTFS* has facilitated the design of algorithms for finding a shorter path according to one or more criteria such as *Raptor* and its derivatives [15]. But also the design of algorithms focusing on accessibility, which is the notion corresponding to the problems of people with disabilities and public transport [18].

GTFS is split into two components. The first one is a static component that includes schedule, fare, and geographic transit information. The second one is a real-time component that includes arrival predictions, vehicle positions and service advisories.

This data format is interesting because, today, it is used by thousands of public transport providers[2]. In addition, in the context of this thesis, recent *GTFS* data are readily available

[1] Definition from https://gtfs.org/
[2] https://gtfs.org/

for the Belgian territory.

- We have *GTFS* data from **STIB-MIVB** agency which allows to define public transport (bus, tram, metro) in the city of Brussels. The data are open and can be downloaded for free.
- We have *GTFS* data from **TEC** agency which is a public transport company operating in the Walloon region of Belgium. The vehicles defined are buses and trams. The data are open and can be easily accessed as well.
- We have *GTFS* data from **De Lijn** agency which is a public transport company operating in the Flemish region of Belgium. Here again the vehicles defined are buses and trams. The data are open and accessible.
- We have *GTFS* data from **NMBS/SNCB** agency which is the Belgian National Railway Company. The type of vehicle defined is train. This agency is available everywhere in Belgium. Like the previous ones, the data are open and accessible.

As we can see, it is easy to find *GTFS* data of the main mobility agencies present in Belgium. Moreover, we also see that this format defines buses, trams, metros as well as trains. However, this format defines much more than that. It defines schedules, routes, stops, transfers, and so on.

Actually, the *GTFS* format is composed of several *.txt* files. Among the many files representing *GTFS* data, some are more important than others and are mandatory if we want our data to be valid. We have listed them here below[3]. As a remark, each file contains required and optional fields.

- `agency.txt` gives information about the agency like his name, url, email, phone number,..
- `stops.txt` describes stops where vehicles pick up or drop off riders. The information given are the name of each stop, its latitude and longitude,...
- `routes.txt` defines transit routes by defining its id,name, type, ... Each route represents a set of trips.
- `trips.txt` defines trips for each route. A trip contains at least two accessible stops during a specific time period.
- `stop_times.txt` defines for each trip the time at which a vehicle arrives and leaves at each stop.
- `calendar.txt` is a conditionally file which describes services. Each service contains dates where it is available and unavailable. Each trip in `trips.txt` is associated to a service.
- `calendar_dates.txt` is also a conditionally file which defines exceptions for the services described in `calendar.txt`.

GTFS Realtime

The purpose of this format is to provide information from agencies in real time. It's an extension to *GTFS* and was designed around ease of implementation, good *GTFS* interoperability and a focus on passenger information. [4]
The *GTFS Realtime* specification currently supports several types of information:

[3]Definitions inspired from: https://developers.google.com/transit/gtfs/reference
[4]https://github.com/google/transit/tree/master/gtfs-realtime/spec/en

- **Trip Updates:** It's about delays, cancellations or changed routes.
- **Service Alerts:** It's about unexpected events that disrupt a station, a route or the entire network.
- **Vehicle Positions:** It's about information about the vehicles including location and congestion level.

Feeds are served via HTTP and updated frequently. For example, the *Stib/MiVB* agency recommends to update feeds each twenty seconds. The data exchange format used is based on *Protocol Buffers*. Beacause it allows the exchange of data directly in binary form, which takes up less space and bandwidth.

As mentioned, *GTFS Realtime* is an extension of *GTFS*, so we need to get information from *GTFS* in order to use *GTFS Realtime* on it.

```
 trip_update {
trip {
  # selects which GTFS entity (trip) will be affected
  trip_id: "1"
}
# schedule information update
stop_time_update {
  # selecting which stop is affected
  stop_sequence: 3
  # for the vehicle's arrival time
  arrival {
    # to be delayed with 5 seconds
    delay: 5
  }
}
```

Above, *GTFS Realtime* will match the parameters `trip_id` and `stop_sequence` with *GTFS* in order to put a delay of 5 seconds.

GTFS-Flex

GTFS-Flex is an extension to *GTFS* whose aim is to model various demand-responsive transport (DRT) services to *GTFS*, which currently only models fixed-route public transport. *GTFS-Flex* is now provides flexible travel plans for public transport through *OpenTripPlanner*.
This extension is interesting in that demand-responsive transport alone accounts for over half of all transit services in the U.S.[5] [10].

GTFS-Flex adds several features such as demand response modelling, continuous stops, route deviation, flags stops, ...[6].
Demand response refers to transportation services in which a vehicle picks up and drops off a rider anywhere within a specified area. Route deviation, on the other hand, describes fixed-route services in which the vehicle may deviate from its route to pick up and/or drop off passengers. Continuous stops refers to services that allow riders to board and/or alight at any point along a route other than fixed stops [10].

In order to make these different features possible and to obtain flexible trips. *GTFS-Flex* modifies and brings new files in *txt* format but also in *GeoJSON* (more discussed at 4.1.2) format.

[5]1 American Public Transit Association. (2019). 2019 Public Transportation Fact Book, 7. Retrieved from https://www.apta.com/wp-content/uploads/APTA_Fact-Book-2019_FINAL.pdf
[6]https://github.com/MobilityData/gtfs-flex/blob/master/spec/reference.md

- `location_groups.txt` refers to multiple pickup/drop-off areas or unordered stops related to a single service.
- `booking_rules.txt` defines whether a service has reservation requirements and provides booking directions.
- `locations.geojson` adds *GeoJSON* locations which describes geographic areas in which demand response services can pick up/drop off riders.
- `stop_times.txt` has fields added to it which refers to booking rules, the time ranges in which a rider can be picked up/dropped off,... [10].

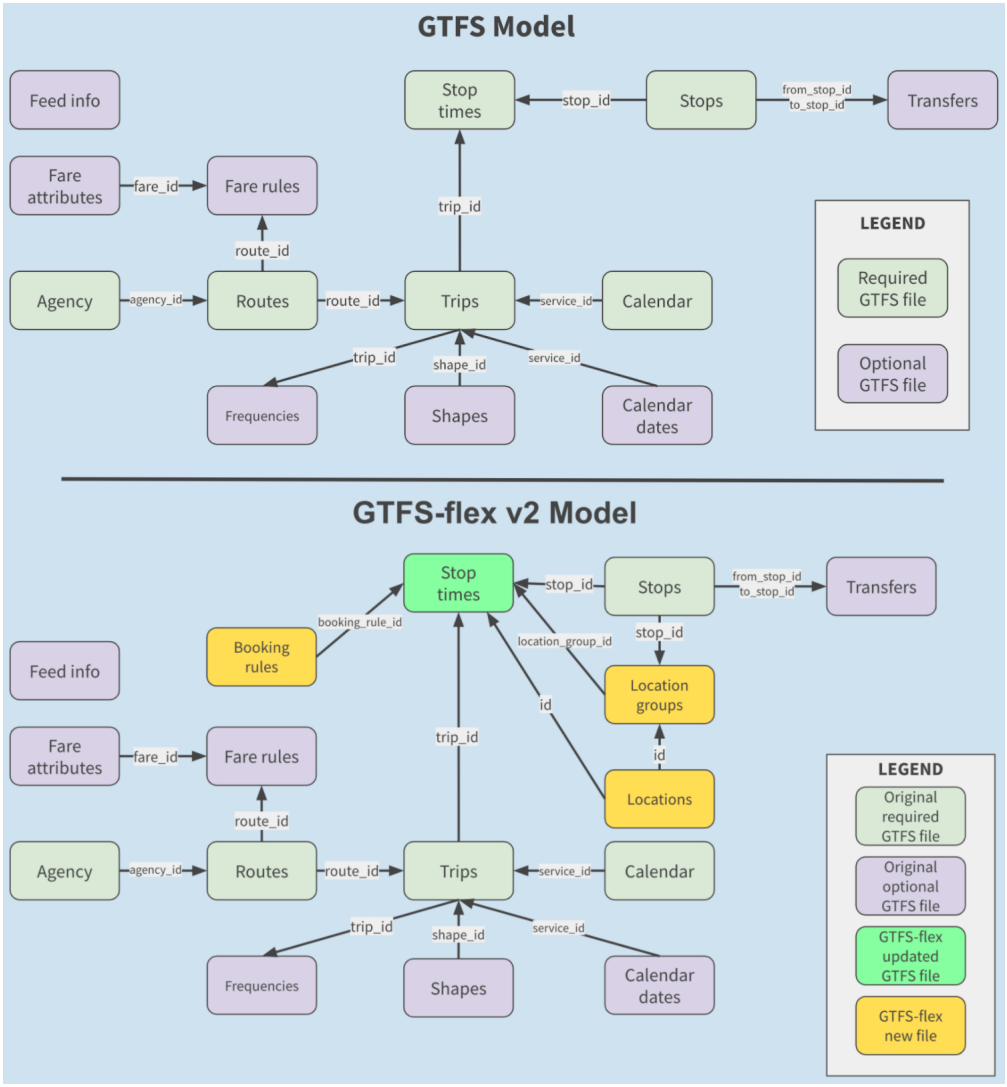


Figure 4.1: Link between *GTFS* and *GTFS-Flex*[7]

4.1.2 GBFS

GBFS, or more concretely, The General Bikeshare Feed Specification, is the open data standard for shared mobility. *GBFS* makes real-time data feeds in a uniform format publicly available online.
GBFS was created in 2014 by Mitch Vars with collaboration from public, private sector and non-profit shared mobility system owners and operators, application developers, and technology vendors. *GBFS* is an open source project. As a consequence, contributors come from across the public sector, the shared mobility industry and everywhere else.

However, one should be aware that this specification tends to meet the following 2 criteria:

- Provide the status of the system at the time of the call
- Do not provide information whose primary purpose is historical

The first version of *GBFS* was launched in 2019. However, developments in the shared mobility industry have pushed the specification to adapt by including new features and capabilities over time. Today (2nd quarter 2022), *GBFS* is at version 2.3. A version 3.0 is currently in development and is planned to be released after the summer 2022[8].

This data format is interesting because it defines today, in a standardised way, many shared mobility vehicles. In the context of this thesis, we collected GBFS data from various operators in the Belgian territory:

- We have the electric scooters of the company **Lime** present in Brussels. These can be borrowed and dropped off anywhere on the public highway (subject to certain restrictions of course).
- We have the bikes of the company **Blue Bike** present in many cities in Belgium. Bicycles can be rented at specific locations (dedicated parking) and must be returned to their original location at the end of the rental period.
- We have the electric scooters of the company **Pony** present in Brussels and Liege. These can be borrowed and dropped off anywhere on the public highway (subject to certain restrictions of course).
- We have the shared bikes and shared electric bikes from the company **Donkey Republic** Present in Kortrijk, a city in Flanders. The principle is similar to that of Villo in Brussels. They can be borrowed from a specific station and dropped off at another station in the city of Antwerp.

As we can see, the *GBFS* specification defines many vehicles, whether they are bikes or shared scooters, whether they are electric or not, whether they have to be dropped off at a specific place or freely in the city. In addition, it can also take into account shared cars, or shared mopeds.
Beyond that, the *GBFS* specification is composed of many files in *JSON* format. These files, in addition to defining the type of vehicle, its motorization and its rental system, define many other specificities.

All *JSON* files contain the same common header information at the top level of the *JSON* response object which is the following:

```
{
"last_updated": 1640887163,
"ttl": 3600,
"version": "2.3",
"data": {
"name": "Example Bike Rental",
"system_id": "example_cityname",
"timezone": "America/Chicago",
"language": "en"
```

[8]https://github.com/NABSA/gbfs/blob/master/gbfs.md

```
}
}
```

where *last_updated* represents the last time data in the feed was updated, *ttl* represents the number of seconds before the data in the feed will be updated, *version* represents the version of *GBFS* currently used and *data* represents data we are going to fetch.

Among the many files that make up this format, we have listed here the most important and those that will be useful for the rest of this work.

- `system_information.json` contains information about system operator, contact info, URL to the operator's website, time zone, ...
- `station_information.json` contains the list of all stations as well as their capacities and locations (in WGS84). It could include the payment methods accepted at each station and if the station supports charging of electric vehicles. This file is required if the system uses docks.
- `station_status.json` contains information about the number of available vehicles as well as docks at each station, each station availability. Like the previous one, this file is required if the system uses docks.
- `free_bike_status.json` gives a description of all vehicles that are currently not rented. It gives their location, the fuel or power battery of each vehicle, the furthest distance in meters that the vehicle can travel with the vehicle's current charge or fuel. But also if the vehicle is already reserved, or if it is unusable (mechanical problem or low battery). This file is required if the system uses dockless vehicles.
- `system_hours.json` describes hours and days of operation when vehicles are available for rental.
- `geofencing_zones.json` describes zones where we are going to apply some restrictions. Besides, it defines the types of vehicle for which we could apply some restrictions. We can, for example, prevent a vehicle from starting and stopping in a certain area, impose a speed limit in a well-defined perimeter, or prevent a vehicle from crossing an area.

The `geofencing_zones.json` file is a little more complex than the others, so we will spend a little more time on it. Actually, the `geofencing_zones.json` contains a *GeoJSON* object called *FeatureCollection*. *GeoJSON* is a specification for storing geographic data in *JSON* format. It supports geometry types like *Point*, *Linestring*, *Polygon*. In addition, you can also find arrays of these types called respectively *MultiPoint*, *MultiLineString* and *MultiPolygon*. Finallly, you can find geometric objects with additional properties called *Feature* and *FeatureCollection* objects.

- GeoJSON definitions[9]
  - **Point**
    - this geometry type is just composed of coordinates

```
{"type":"Point",
"coordinates":[4.34,52.88] }
```

[9]Definitions taken from: https://datatracker.ietf.org/doc/html/rfc7946#section-3.1.2

    * **MultiPoint** is an array of *Point*

- **LineString**
    * this geometry type is composed of single or multiple *Points*

```
{"type":"LineString",
"coordinates":[ [4.34,52.88], [4.31,53.12],
[4.30,53.45], [4.29,54.00] }
```

    * **MultiLineString** is an array of *LineString*

- **Polygon**
    * this geometry type is composed of single or multiple *LineString*

```
{"type": "Polygon",
"coordinates": [
[[35, 10], [45, 45], [15, 40], [10, 20], [35, 10]],
[[20, 30], [35, 35], [30, 20], [20, 30]]
] }
```

    * As you could see, the polygon above is composed of two *LineString*. Each of those lines has the same beginning and end thus it forms a ring. The first ring represents always the outer ring while the other rings, here the second one, represent the inner rings. Hence you can represents a island and a lake inside it for example.

Figure 4.2: Two polygons composed of two rings

    * **MultiPolygon** is an array of *Polygon*

- **Feature**
    * this object is composed firstly, of a geometry type as well as properties which are applied to it.

```
{"type":"feature",
"geometry":{ "type":"Point", "coordinates":[4.34,52.88] },
"properties":{
"type": "Pizza restaurant",
"name": "Aqua E Farina",
"phone_number" :+32XXXXXX}}
```

    * **FeatureCollection** is an array of *Feature*

The `geofencing_zones.json` contains a *FeatureCollection* of *MultiPolygon*. So we have several sets of locations in which we apply certain properties (Can we start or stop the vehicle at this location? Can we cross this location with the vehicle? ...). However, the data and their accuracy obviously depend on the different operators.
Figure 4.3 shows locations (polygons) in which properties have been defined within the city of Brussels (data provided by **Pony**).

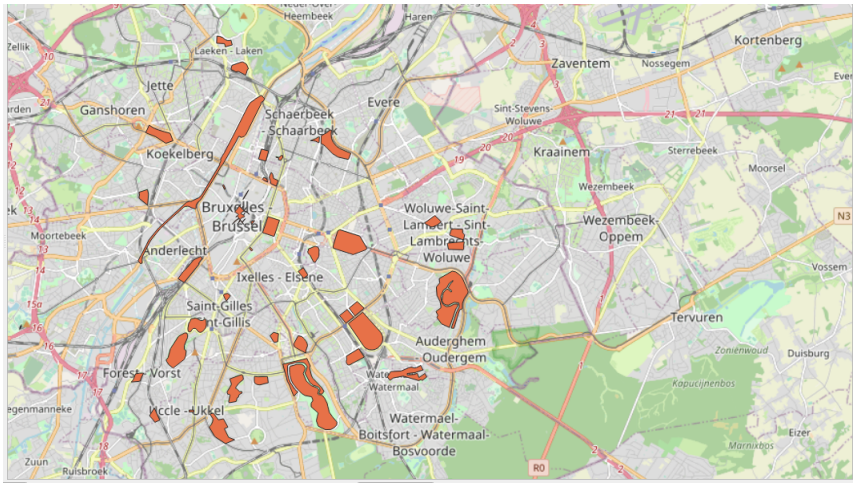


Figure 4.3: Vizualisation a part of the file *geofencing_zones.json* with QGis

Figure 4.4 shows an area in which restrictions have been applied (data provided by another Belgian operator).

Figure 4.4: Vizualisation a part of the file *geofencing_zones.json* with QGis

We see that the boundaries of the covered area appear to be significantly less precise than those of the other areas of the **Pony** operator. Indeed, the boundaries cover part of the sea and the Netherlands. This is due to the fact that the *Polygon* is defined by fewer *LineStrings* and *Points*. As a result, we lose precision and behaviours like this can occur. Unfortunately this accuracy depends on the data provided.

4.1.3 NeTEx/Siri

NeTEx (Network Timetable Exchange) defines a standard for exchanging public transport passenger information data in *XML* format.
Data in *NeTEx* format is encoded as *XML* documents that must conform exactly to the schema. The schema can also be used to create bindings to different programming languages, automating part of the implementation process for creating software that supports *SIRI* formats [3].

Regarding transmission, documents in *NeTEx* format are computer files that can be exchanged by a wide variety of protocols (http ftp, email, portable media, etc). In addition, a *SIRI* based protocol is specified for use by online web services. The common *SIRI* framework is used to describe a specific *NeTEx*/data service (SIRI-NX) with specialized messages that can be used to request and return messages containing data in *NeTEx* format, as well as publish/subscribe messages for push distribution. The SIRI-NX responses return a *NeTEx XML* document that

satisfies the request criteria (and also conforms to the *NeTEx* schema) [3].

NeTEx is divided into several parts. It consists of 5 parts so far:

- *Part 1* refers to the fixed Network (stops, routes, lines, etc.).
- *Part 2* refers to the timetables.
- *Part 3* refers to the fares data.
- *Part 4* refers to information relevant to feed passenger information services and excluding operational and fares information.
- *Part 5* refers to alternative modes such as car sharing, cycle sharing, car rental, cycle rental as well as carpooling.

NeTEx refers to static data while *SIRI* is oriented to real-time information[10].

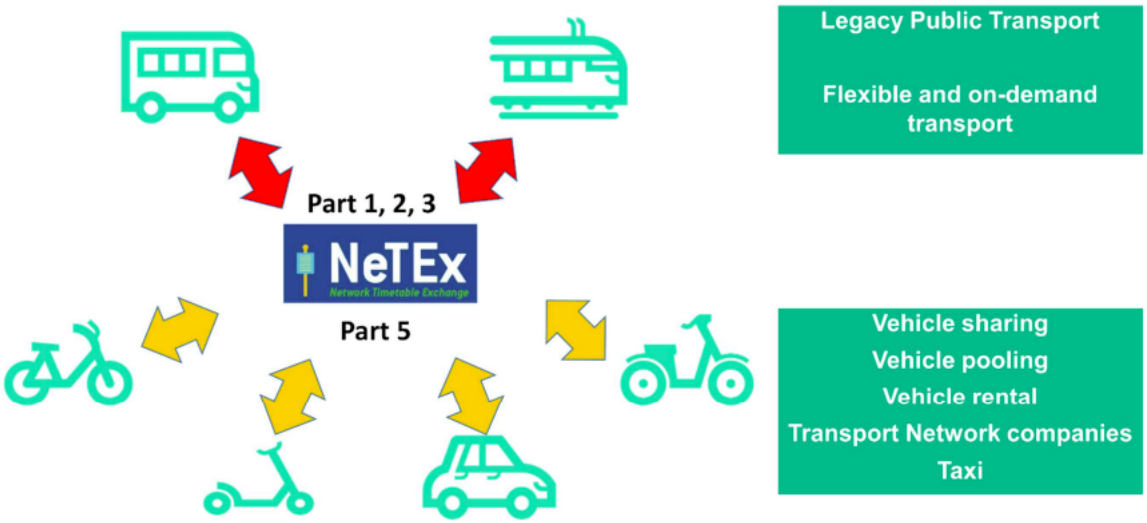


Figure 4.5: Overview of NeTEx[11]

With the recent addition of these new modes of transport, *NeTEx* has had to adapt by adding a few attributes and concept to the existing ones. We can mention for example the enhancement of booking informations in order to cover new modes of transport as well as the enhancement of *NeTEx* routes, vehicles profiles or fare offers,... Besides new concepts are added such as fleet of vehicles, vehicle profiles,... [1].

As a remark generate a full *GTFS* data set from *NeTEx* is possible but not vice versa[12].

In addition, there are also similarities between the *GBFS* format and the last part of *NeTEx* implemented so far. A detailed mapping has been started with the latest *GBFS* version[13].

[10]https://netex-cen.eu/?page_id=534
[12]https://netex-cen.eu/?faq=how-does-netex-compare-with-gtfs
[13]https://www.netex-cen.eu/wp-content/uploads/2021/05/Status-of-MMTIS-standards.pdf

Figure 4.6: Mapping GBFS to NeTEx/SIRI [14]

The last release of *NeTEx* is 1.2 and was released by the end of March 2022. It defines schemas for *Part 1*,*Part 2*,*Part 3* and *Part 5*.[15]

[15]Github: https://github.com/NeTEx-CEN/NeTEx

Chapter 5

Multi-Modal Routing Tools

5.1 Multi-Modal Routing Tools

As for single-modal routing, many tools, open-source or not, have been developed for multi-modality.

5.1.1 OpenTripPlanner

OpenTripPlanner is an open source software project written in java that provides passenger information and transportation network analysis services. It finds itineraries combining transit, pedestrian, bicycle, and car segments through networks built from widely available, open standard *OpenStreetMap*.[1]

The project started in 2009 and it continues to be updated. Indeed the second major version of *OpenTripPlanner*, called *OpenTripPlanner 2* has been under development since 2018 and was released in November 2020. The project has many contributors, over 12,000 commits and has been forked 900 times. This makes it the largest multi-modal open source project to date.

Even though *OpenTripPlanner 1* is widely used, its transit routing approach is obsolete. There exists several more ressource-efficient approaches. Besides it has also accumulated large amounts of experimental code and specialized tools, which complicate long-term maintenance.

OpenTripPlanner 2 offers much better performance[2] in larger transportation networks and geographic areas, and a wider variety of alternative itineraries. It's public transit routing component has been completely rewritten, and is now distinct from bike, walk, and motor vehicle routing.

Although it's is not a complete replacement because there are some use cases where *OpenTripPlanner 1* will be better suited. We will use *OpenTripPlanner 2* as a basis for understanding the structure of the network used and the routing algorithms.

In addition, this new version support new data formats. For example, the user can now provide *NeTEx*, *GTFS*, *GBFS* or real time data such as *GTFS-Realtime* and *Siri*.

[1] This definition is taken from: https://www.opentripplanner.org/
[2] http://docs.opentripplanner.org/en/latest/Version-Comparison/

Graph representation

In *OpenTripPlanner 2* ,the pedestrian network and the transit network are created separately. The design of the pedestrian network is the same as in the first version of the tool[3]. In the pedestrian network, nodes represent intersections and edges represent road segments between intersections. As the edges are directed, the road segments have 2 edges, each going in an opposite direction (except in special cases).

As its routing algorithm Raptor is not based on Dijkstra, it is not necessary to represent the transit network as a graph. Nevertheless, we can note the presence of nodes representing stops as well as "pre-board" edges linking street network nodes to these nodes. These pre-board edges represent the constraint to access a public transport and their cost can be modified by the user (depending on his will or not to take public transport).

Routing algorithms

. Walk-only and bicycle-only trips are generally planned using the A* algorithm with a Euclidean heuristic, this approach is the same as *OpenTripPlanner 1.* Walk + transit as well as bike are planned by A* with Tung-Chew heuristic [34].

Regarding the transit routing algorithm, *OpenTripPlanner 2* implements several *Raptor* algorithms such that the simple *Raptor* (see 3.2.2), the *Range Raptor* (see 3.2.2) as well as a *Raptor* based algorithm. More specifically, it implements the *Range Raptor* algorithm with *Multi-criteria pareto-optimal* search, namely *Multi-criteria Range Raptor (McRR).*
McRR search aims to return the Pareto-optimal set of paths considering the following optimisation criteria:

- *arrival time*
- *number of transfer*
- *generalized cost* : a function that includes *waiting time, walking distance, operator, travel-time...*

These 3 implemented algorithms allow the user to make their own choices. For the sake of speed, the user may prefer a simple *Raptor* or RR search instead of McRR. Even though McRR allows a true support for multi-criteria search, these performances are quickly affected.
For the moment, McRR manages 3 criteria, namely the *number of transfers*, *the arrival time* and *the generalized cost*. The developers plan to exclude *the waiting time* or *the operator* from *the generalized cost* and to put them as criteria.
Several tests were carried out, and it was found that RR could take up to 80 ms while McRR could take up to 400 ms for the same configuration. Adding the *walking time* as a criterion could increase the test time to 1000 ms.[4]

Features

The tool offers a web client that can be accessed locally. To which OSM data, *GTFS* data as well as three *JSON* configuration files can be fed. In these 3 configuration files, 2 are important, namely `build-config.json` and `router-config.json`.
The first one describes the user preferences which cannot be modified without rebuilding the graph.

[3]Structure of the graph detailed: https://github.com/opentripplanner/OpenTripPlanner/wiki/GraphStructure
[4]Performances tests and more: https://github.com/opentripplanner/OpenTripPlanner/issues/2626

The second describes the preferences that can be modified in run-time. Among these preferences, we can change the default speeds (walking, driving, ...) or modify some returned results. What makes the tool interesting for multi-modal routing is that it allows the inclusion of *GTFS* data (or *NeTEx* data) from transport agencies. It also supports *GTFS-RT* data which takes into account possible traffic contingencies. In addition, it can fetch data from some bike rental agencies such as *JCDecaux* (Villo) or fetch data in *GBFS* format.

In Figure 5.1, you can see the trips proposed by the web client, in transit mode, taking the OSM data from Brussels and the *GTFS* data from the STIB. The application returned 29 trips. These proposed trips depend on the preferences that the user has provided in `router_config.json`. In our case, we wanted the "best" trips between 3pm and 3:30pm. The tool therefore ran the Range Raptor algorithm to return the Pareto-optimal paths.

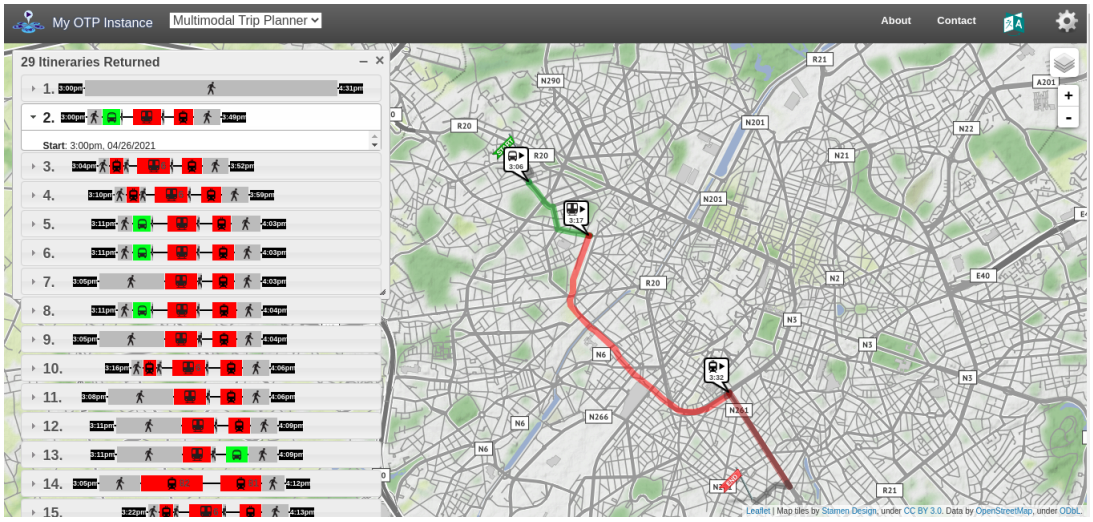


Figure 5.1: Several itineraries using *GTFS* returned by *OpenTripPlanner*

In Figure 5.2 you can see two trips, still in Brussels, using *Villo* shared bikes. Although they start and end at the same destination, the two trips are different because the user has changed his preferences for travelling by bike (see the triangle at the bottom left).

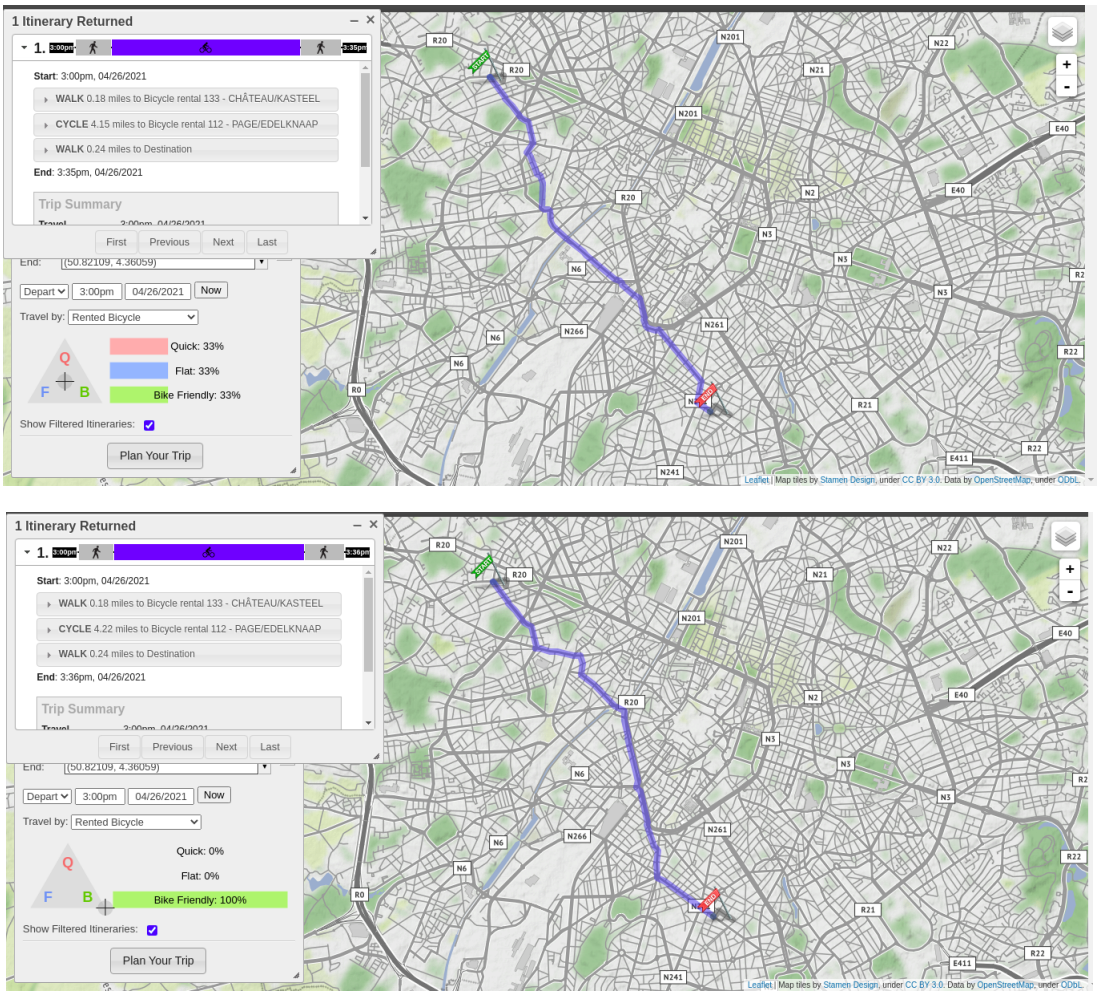


Figure 5.2: trips using bike and user preferences

As mentioned earlier, a choice has to be made on each road segment. Indeed, it has to be determined whether a car, a bicycle or a pedestrian can move on the road segment (edge). The developers of *OpenTripPlanner* also had to make this choice [5]. In addition, thanks to the web client, the user can see, via a choice of colours, the different types of transport permitted on each segment (which can be useful in the debugging phase).

Figure 5.3: *traversal permissions* parameter enabled on *OpenTripPlanner*

[5] http://docs.opentripplanner.org/en/latest/Troubleshooting-Routing/

The colours in Figure 5.3 represent:

- grey lines: all
- red lines: car only
- blue lines: pedestrian + bike
- green lines: pedestrian only
- yellow lines: links from the street graph to the stops

5.1.2 Attempted Partioning

An attempt to partion *OpenTripPlanner*, called *OpenMove*[6], was made a few years ago. The idea was to partition the graph into several regions, each containing a *GTFS* dataset. Then, thanks to a new distributed search engine, these servers could cooperate to find a route.

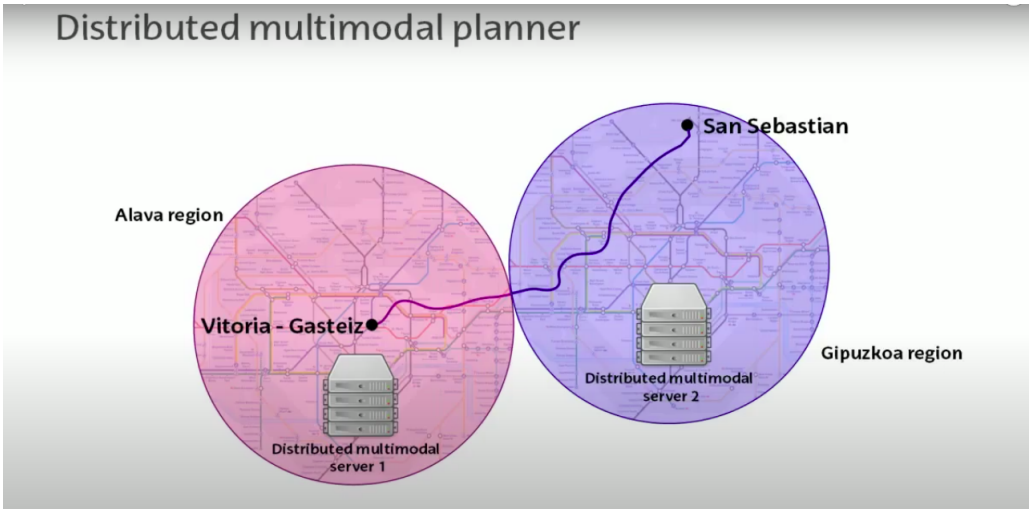


Figure 5.4: image from a video presentation of OpenMove[7]

Unfortunately, the project is based on an old version of *OpenTripPlanner* (version 0.9) and has not been updated since[8].

5.2 Other tools

Many companies related to intermodality exist and adapt their routing engine and graph builder according to the evolution of the state of the art.

For example, *Jeasy* [9] is a Belgian start-up founded in 2018 that is essentially based on multimodal. It is available throughout Belgium and offers its users the following modalities:

- personal mobility (own car, own bike, own scooter)
- Shared mobility (shared cars, shared bikes, shared scooters)

With regard to user preferences, *Jeasy* is currently somewhat limited. The user can only choose which type of vehicle he would like to use to make a journey.
Regarding its business plan, the application mainly focuses on the CO2 consumption avoided

[6]https://www.open-move.org/home/
[8]Github: https://github.com/solidrockit/OpenMove
[9]https://jeasy.ai/

by using these alternative means of transport. By using the application, the user will earn *EcoMiles* corresponding to the CO2 he or she has saved compared to the same journey made with his personal car. These *EcoMiles* then allow him to access content from companies in partnership with *Jeasy*.

Google Maps, on the other hand, offers its users the choice between a journey with their own car or bicycle, probably for reasons of ease, or to join the transit network on foot (bi-modal). With the emergence of new means of transport, *Google Maps* has started to collaborate with the various players in the market and offers alternative bi-modal routes to its users (walk + shared bike, walk + shared car, ...).

Chapter 6

Case Studies

6.1 Case studies

As mentioned above, *OpenTripPlanner* is a comprehensive tool, supporting many data formats. For the purpose of this thesis, we will focus on the *GTFS* and *GBFS* formats, the latter allowing to define the set of shared mobility vehicles.
As a result, we will perform tests with on some classic and some more particular cases to determine whether these formats are correctly supported. The idea is to highlight flaws in the algorithm's ability to provide consistent trips.

It should be noted that some faults/errors of the tool were observed during this test phase (and being external to the good management of the formats by the tool)

6.1.1 Car Tests

The trips calculated with the cars are mostly limited. *OpenTripPlanner* is oriented toward public transit routing. Therefore, if you travel by car-only mode, no alternative route will be offered. [1]

Traffic jam

OpenTripPlanner does not offer a real-time traffic display, so OTP is not able to calculate a route that would avoid an accident or traffic jams. However, public traffic data are available. For example, the city of Brussels calculates road traffic using cameras scattered around the capital and allows access via an API[2]. Several formats are available such as *Json* or *CSV*.
Other open-source alternatives such as *open traffic*[3] exist. The latter is a global data platform that stores vehicle positions using users' smartphones. This makes it possible to obtain traffic in real time but also to make statistics.

Restrictions

OpenTripPlanner do not take account of restrictions zones like *Low Emission Zone (LEZ)*. These restrictions are found in Belgium, France, Denmark, Germany, ...[4]
Actually, these restrictions are defined on *OpenStreetMap* maps via a tag called *boundary* and its value *low_emission_zone*[5]. It would therefore be relatively easy to adapt *OpenTripPlanner* so that it warns the user if the calculated trip passes through a restricted LEZ.

[1] https://groups.google.com/g/opentripplanner-users/c/TYTlcYZN7UY/m/pgyIvA28BwAJ
[2] https://data.mobility.brussels/traffic/api/counts/
[3] https://github.com/opentraffic/otv2-platform
[4] Exhaustive list can be found here: https://en.wikipedia.org/wiki/Low-emission_zone
[5] https://wiki.openstreetmap.org/wiki/Tag:boundary%3Dlow_emission_zone

6.1.2 GTFS Tests

In order to prove the reliability of the results provided by *OpenTripPlanner* (based on *GTFS*) we will perform several special case tests to see if this format is correctly handled by the tool. Next, we will provide it with several *GTFS* data and test whether the results returned are consistent.

We are going to use a subset of *GTFS* data from Stib to make the tests. Firstly, we put the `stib-gtfs.zip` in the Datas/ folder and add its name in the `build-config.json` file like this.

```
{
"storage": {
  "localFileNamePatterns": {
     "gtfs" : "stib-gtfs.zip" }
   }
}
```

Services

In this test, we will mainly use the two `calendar_dates.txt` and `calendar.txt` files in order to check if a service is correctly handled by the tool. Firstly we have defined the service `200039050` as being active on weekdays. This service represents trips by metro. To do this, in the `trips.txt` file, we need to associate each metro trip with this service.

```
service_id,monday,tuesday,wednesday,thursday,friday,saturday,sunday,
start_date,end_date
...
200039050,1,1,1,1,1,0,0,20221028,20221031
...
```

calendar.txt

Then we associate trips to our service.

```
route_id,service_id,trip_id,trip_headsign,direction_id,block_id,shape_id
...
1,200039050,1,ERASME,1,8228361,005m0129
1,200039050,2,HERRMANN-DEBROUX,0,8228361,005m0077
...
```

trips.txt

Note that the service is defined as active from 10/28/2022 to 10/31/2022 Figure 6.1 shows a itinerary computed using a metro line on Friday 28 October 2022.

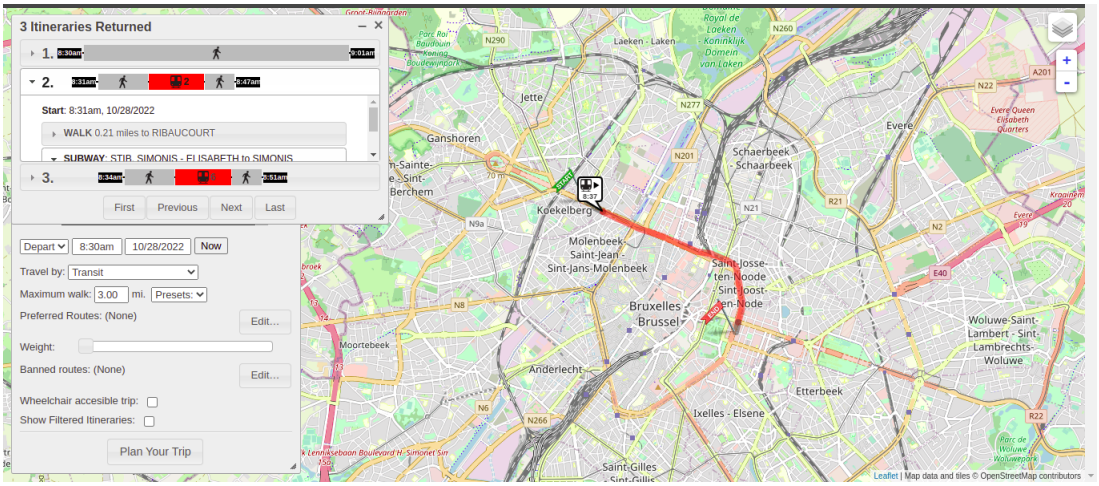


Figure 6.1: Trip computed for our defined service

However, we will now make an exception to our previously defined service by making the service inactive on Friday 28 October 2022. To do this, we add this line to our `calendar_dates.txt` file.

```
service_id,date,exception_type
...
200039050,20221028,2
...
```

calendar_dates.txt

We notice that the `exception_type` parameter is set to `2`. This means that service has been removed for the specified date.

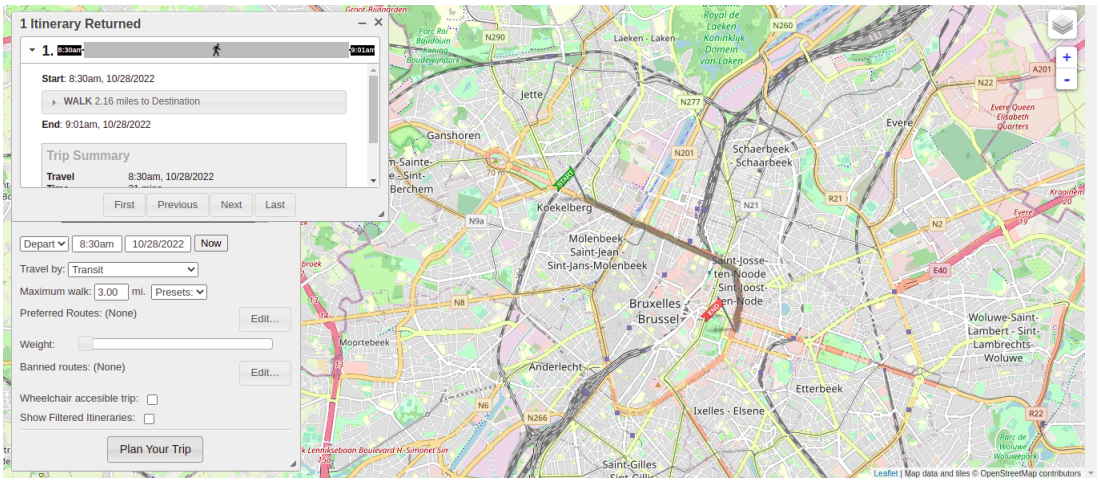


Figure 6.2: Trip computed with a removed service

As the service associated with the various metros in the city is inactive, the journey planner has found an alternative itinerary as shown in Figure 6.2.

Bike and Transit

Public transport such as train or metro offers passengers the possibility to take their own bike in the vehicle. It is in `trips.txt` file that the optional parameter `bikes_allowed` is defined. We set this parameter to `1` and then, we try to compute a trip combining our bike and transit network.

```
route_id,service_id,trip_id,trip_headsign,direction_id,block_id,
shape_id,bikes_allowed
```

```
....
1,200039050,1,ERASME,1,8228361,005m0129,1
1,200039050,2,HERRMANN-DEBROUX,0,8228361,005m0077,1
...
```

trips.txt

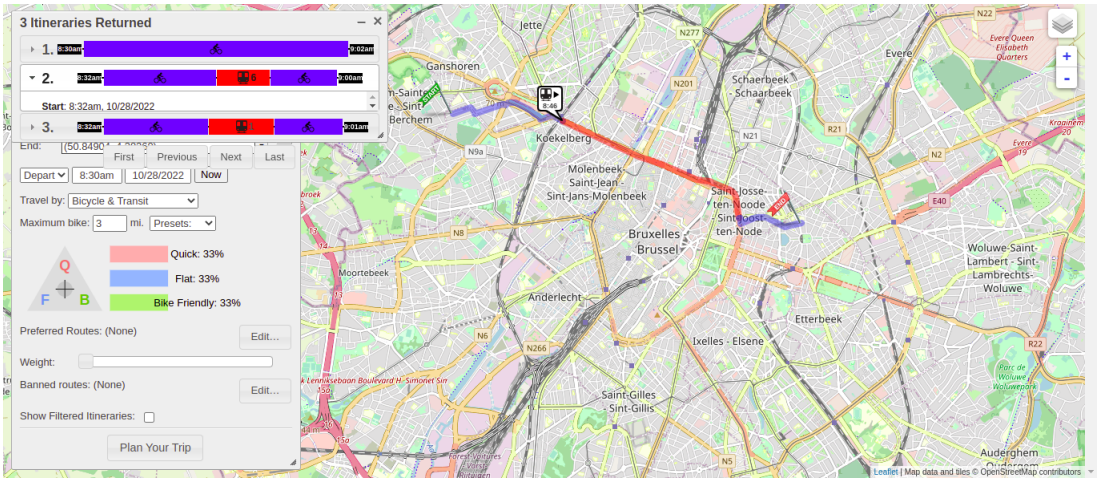


Figure 6.3: Trips computed with personal bike and transit service

Several routes are given, some of which combine public transport and our own bike. As a result, it turns out that *OpenTripPlanner* handles this combination correctly (as shown in Figure 6.3).

GTFS combination

A traveller often needs to make one or more transfers. It is therefore necessary to carry out some tests by combining different means of transport. As a consequence we will simulate a few journeys with a few transfers using *GTFS* data.

Several routes were carried out, combining transfers within a single *GTFS* dataset (transfers involving a single agency), but also transfers involving several *GTFS* datasets (several agencies). Moreover, the trips were made within the city of Brussels but also in Wallonia (the train allowing to easily connect the 2 regions).
It should be noted that no inappropriate behaviour of *OpenTripPlanner* was observed. The routes returned were similar to those returned by an application like *Google Maps*, or by the Stib application.

Figure 6.4 shows a journey from Brussels to Mons combining several *GTFS* data (bus, tram, train).

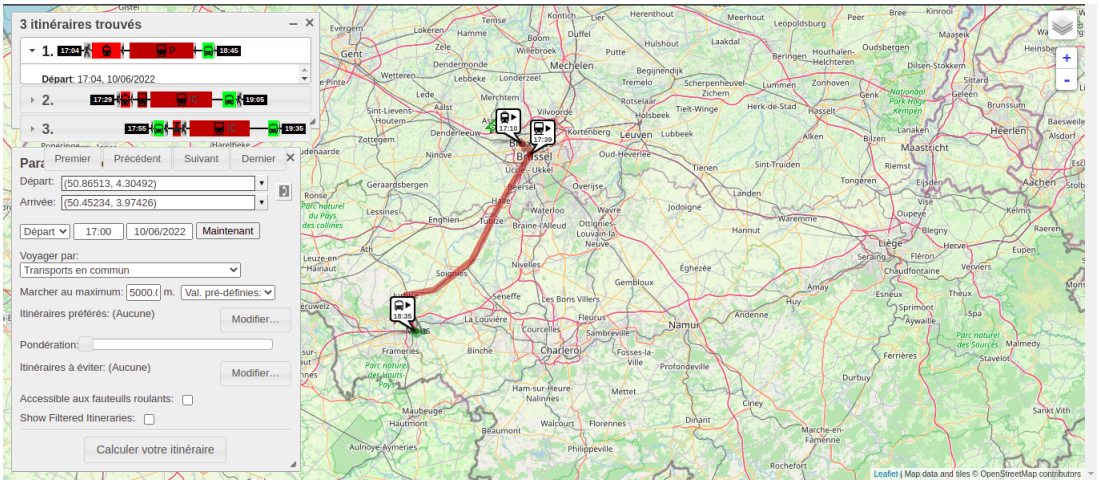


Figure 6.4: WebApp from OpenTripPlanner computing a trip combining several GTFS data

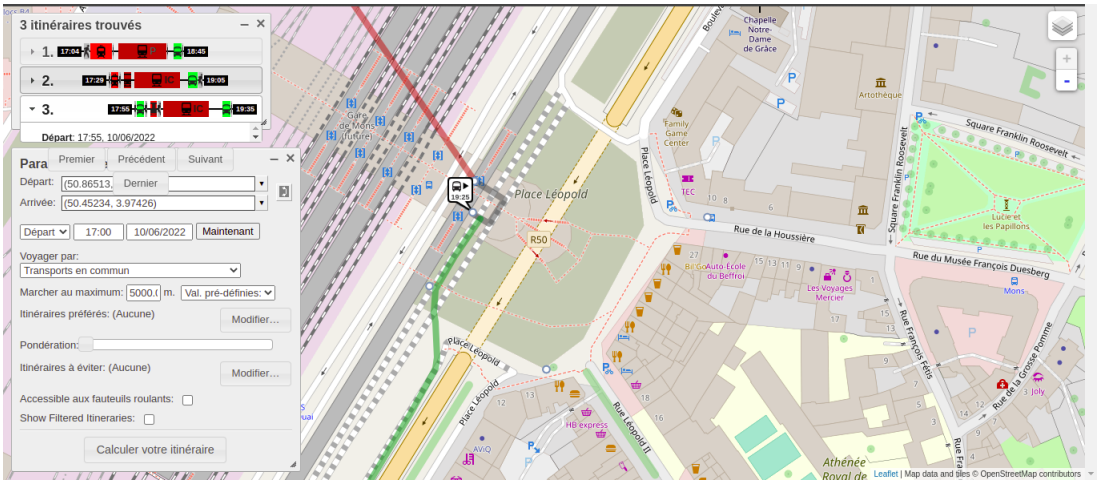


Figure 6.5: OpenTripPlanner WebApp focusing on a particular transfer

We can activate the debugging phase to better understand how the transfer works (see Figure 6.6). And we observe that the user switches from one network to another via yellow segments provided for this purpose (as explained in 5.1.1).

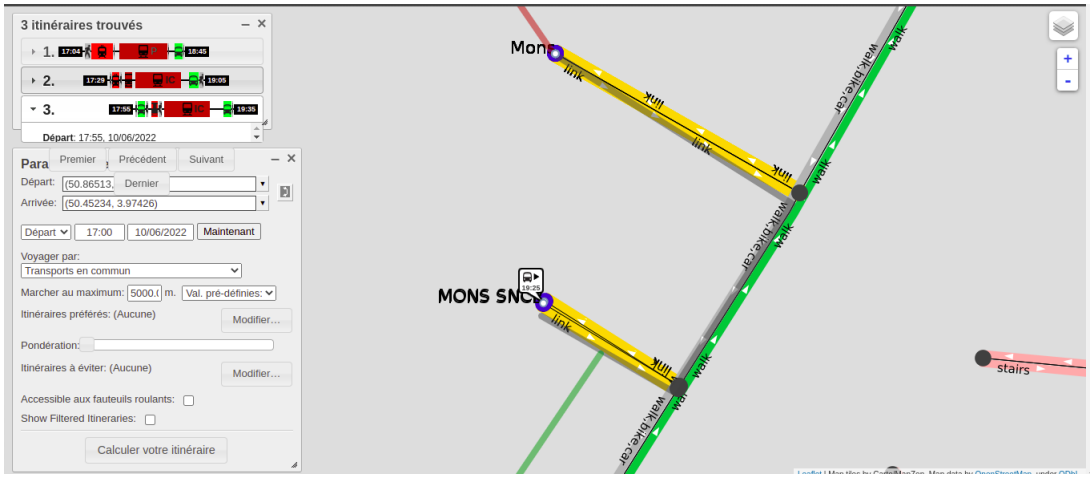


Figure 6.6: Yellow links connect pedestrian network to transit networks

6.1.3 GBFS Tests

As mentioned earlier, the *GBFS* format covers a multitude of shared mobility vehicles. Since version 1 of *OpenTripPlanner*, the *GBFS* format is supported. In addition, version 2.0 of

OpenTripPlanner supports floating vehicles (which means dropping off your vehicles, not at a specific station, but anywhere in the city). All you have to do is set the `FloatingBike` and `APIBikeRental` parameters in `otpFeatures.json` to true.

In the `router-config.json` file, there is an object `updater` including several fields, so the `url` field allowing us to contact an api to get the *GBFS* data.
Here is an example using the Lime API:

```
...
 "updaters": [
{
"type": "bike-rental",
"sourceType": "gbfs",
"frequencySec": 60,
"url": "https://data.lime.bike/api/partners/v2/gbfs/brussels/gbfs.json",
"language": "en",
"allowKeepingRentedBicycleAtDestination": true
}
]
```

router-config.json

We will now perform a series of tests to see if *OpenTripPlanner* supports this format correctly and does not provide routes that are wrong, biased for some reason.
To do this, we are going to use a Apache Web server in a local machine and use it to contain *GBFS* data that we will generate. The *GBFS* data generated will contain one vehicle, located in Brussels, that we will use to execute our tests.
Then we will make these tests:

- Try to use a vehicle to make a trip despite the fact that the vehicle is already reserved by someone else.
- Try to use a vehicle to make a trip despite the fact that the vehicle cannot be used for some reason.
- Try to use a vehicle to make a trip composed of a few kilometres despite the fact that the vehicle has not sufficient autonomy.
- Try to drop a vehicle off in a location that prohibits it.
- Try to pass through an area that prohibits this type of vehicle.
- Try to take a vehicle at a certain time despite the fact that the company has prohibited any use at that time.

The necessary files in order to make these tests are the following:

- `gbfs.json`
- `system_information.json`
- `free_bike_status.json`
- `vehicle_types.json`
- `geofencing_zones.json`
- `system_hours.json`

- `station_information.json`
- `station_status.json`

The content of the files required for each test will be detailed in the following sections.

Vehicle reserved

In order to check if the tool manages correctly vehicles already reserved, we will modify the parameter `is_reserved` in the `free_bike_status.json` file.

```
{"last_updated":1654101931,
"ttl":60,
"version":"2.2",
"data":
   {"bikes":
      [{"bike_id":"BIKE-TEST","lat":50.86678,"lon":4.29844,
      "is_reserved":true,"is_disabled":false,
      "current_range_meters":3256,"vehicle_type_id":"2",
      "last_reported":1654101923,
      "vehicle_type":"scooter"}]
   }
}
```

free_bike_status.json

Regardless of our modifications, Figure 6.7 shows that *OpenTripPlanner* still offers to use our vehicle to make a trip, even if it is already reserved by someone else.

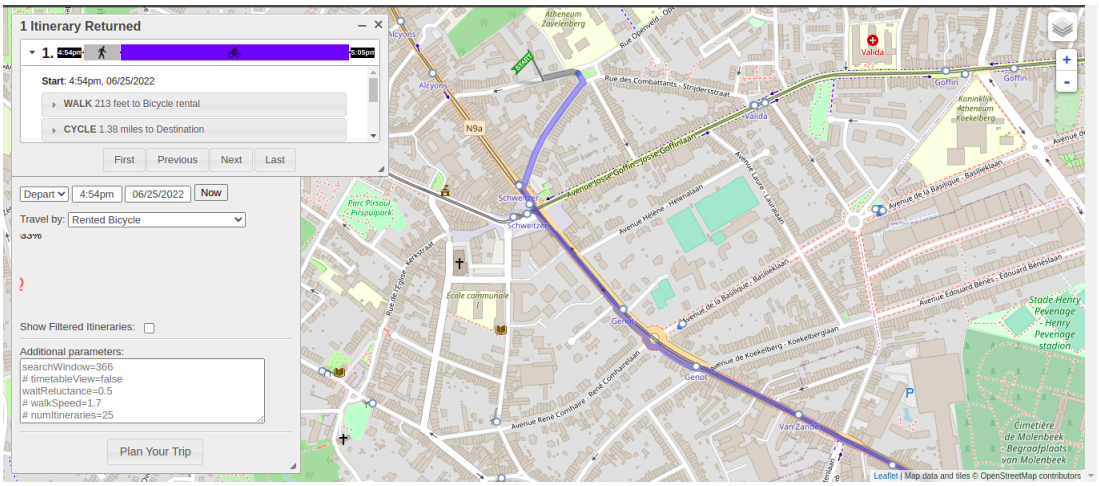


Figure 6.7: Trip computed with already reserved shared scooter

Vehicle disabled

A vehicle may have a mechanical problem or a low load. It is then noted as disabled by the *GBFS* format. This vehicle can therefore not be used to make a trip. Let's take our test vehicle and modify the `is_disabled` parameter.

```
{"last_updated":1654101931,
"ttl":60,
"version":"2.2",
"data":
   {
   "bikes": [{"bike_id":"BIKE-TEST","lat":50.86678,"lon":4.29844,
   "is_reserved":false,"is_disabled":true,
   "current_range_meters":3256,"vehicle_type_id":"2",
   "last_reported":1654101923,
```

```
    "vehicle_type":"scooter"}]
    }
}
```

free_bike_status.json

It turns out that *OpenTripPlanner* still offers us to take this vehicle for a trip. Although it is unable to do so according to the data provided.

Vehicle with a low battery

In order to do this test, we will set the `current_range_meters` parameter to 500. As a result, the vehicle can travel a maximum of 500 metres before being unloaded. The test will consist of to drive the vehicle for several kilometres.

```
{
"last_updated":1654101931,
"ttl":60,
"version":"2.2",
"data":
     {
     "bikes": [{"bike_id":"BIKE-TEST","lat":50.86678,"lon":4.29844,
     "is_reserved":false,"is_disabled":false,
     "current_range_meters":500,"vehicle_type_id":"2",
     "last_reported":1654101923,
     "vehicle_type":"scooter"}]
     }
}
```

free_bike_status.json

It is important to note that our vehicle has been defined as fully electric in `vehicle_types.json` file.

```
{
"last_updated": 1609866247,
"ttl": 60,
"version": "2.2",
"data": {
   "vehicle_types": [{
      "vehicle_type_id": "2",
       "form_factor": "scooter",
       "propulsion_type": "electric",
       "name": "TEST_Scooter"
    }]
     }
}
```

vehicle_types.json

Unfortunately, once again, the result received is not the desired one. Indeed, Figure 6.8 shows that *OpenTripPlanner* proposes us to drive several kilometres with our vehicle.

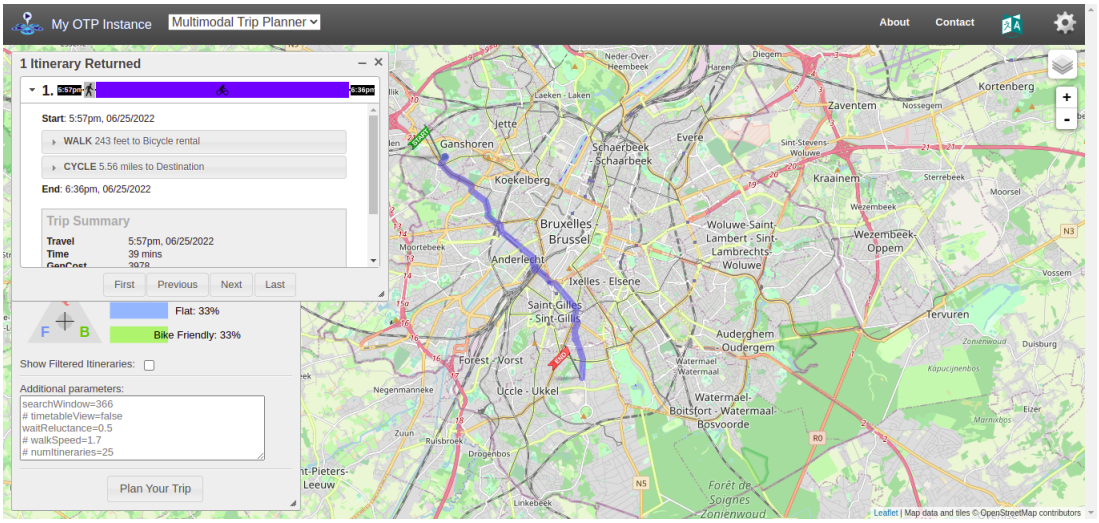


Figure 6.8: Long trip with a too low battery vehicle

Dock systems

The following tests will focus on vehicles that are located at specific stations. The idea, in the first instance, is to determine whether a vehicle can be borrowed from an empty station. And in a second step, the idea is to determine whether a vehicle can be returned to an already full station.

To do this, we will modify the parameters of the `station_information.json` and `station_status.json`. First, we define two stations we will use for the tests with the following parameters.

```
{
"last_updated": 1657378899,
"ttl": 60,
"version": "2.2",
"data": {
   "stations": [
      {
       "station_id": "1",
       "name": "Station FROM",
       "lat":50.86678,
       "lon": 4.29844,
       "capacity": 1
      },

      {
       "station_id": "2",
       "name":"Station TO",
       "lat": 50.84690,
       "lon": 4.36247,
       "capacity": 1
      }
    ]
    }
}
```

station_information.json

The parameter `capacity` represents the total number of vehicles that can be contained within this station. The `station_status.json` file will give us more information about each station.

```
.....
  "stations": [
    {
     "station_id": "1",
```

```
      "num_bikes_available": 1,
      "vehicle_types_available": [
        {
         "vehicle_type_id": "2",
         "count": 1
        }
      ],
      "is_installed": true,
      "is_renting": true,
      "is_returning": true,
      "last_reported": 1657408285,
      "num_docks_available": 1
    },
    {
      "station_id": "2",
      "num_bikes_available": 0,
      "vehicle_types_available": [
       {
        "vehicle_type_id": "2",
        "count": 0
       }
      ],
      "is_installed": true,
      "is_renting": true,
      "is_returning": true,
      "last_reported": 1657408490,
      "num_docks_available": 1
     }
    ]
  }
}
```


station_status.json

Other interesting parameters as `is_renting` and `is_returning` respectively specify that it is possible to rent a vehicle from this station and that it is possible to return a vehicle to this station. If these parameters are set to `false`, it means that the station is temporarily taken out of service.
Figure 6.9 shows a trip from our "Station From" to our "Station To" in if both stations are accessible.

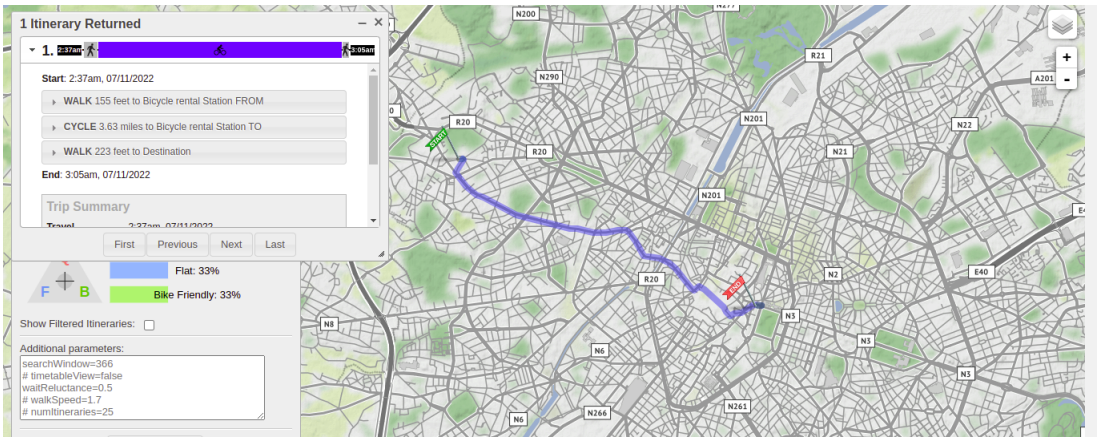


Figure 6.9: Trip returned with our vehicle in a normal situation

- Now we change the `num_bikes_available` parameter to `0` so that there are no vehicles available at the starting station and we restart the computation.
- Then we change the `num_docks_available` parameter to `0` so that there are no docks available at the ending station and we restart the computation.

Figure 6.10 shows the journey computed is different. There are no accessible stations, so the journey is made on foot.

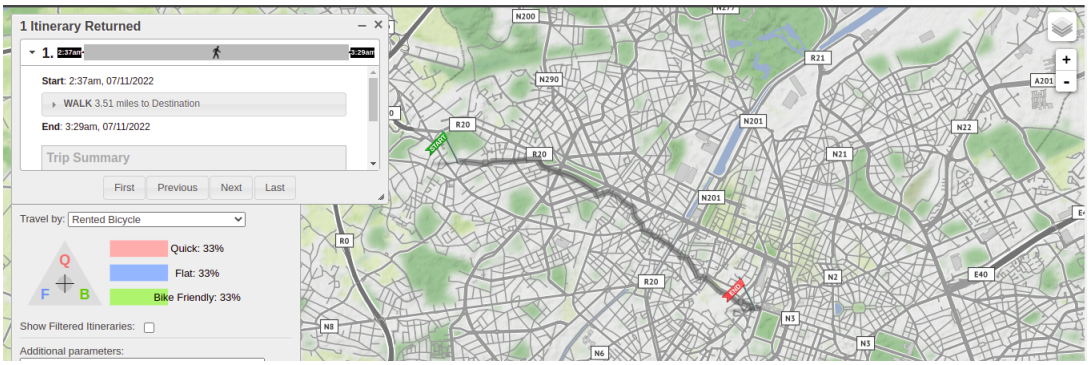


Figure 6.10: Trip returned when no vehicle is available

We see that this part of *GBFS* is correctly handled by *OpenTripPlanner* and prevents the user from getting biased results. However, we can go further and look at the behaviour of our tool if we indicate that one of our 2 stations is out of service.

- we change the `is_renting` parameter to `false` so that our starting station is out of service and we restart the computation.
- we change the `is_returning` parameter to `false` so that our ending station is out of service and we restart the computation.

Unfortunately, in both cases, the proposed trip contains the vehicle present in our departure station.
Once again, we observe a partial management of the *GBFS* format by *OpenTripPlanner*.

Time restrictions

Some operators offer restrictions on the rental of their vehicles. As explained above, the `system_hours.json` file describes the hours and days when vehicles are available for rental. And therefore the hours and days when the service is not available.

For our part, our test operator will define the service as being available on weekdays and therefore unavailable at weekends. Then we will push our test further by making the service available only at certain hours of the day.

```
{
"last_updated": 1609866247,
"ttl": 86400,
"version": "2.2",
"data": {
"rental_hours":[
       {
        "user_types": [ "member", "nonmember" ],
        "days": ["mon", "tue", "wed", "thu", "fri"],
        "start_time": "00:00:00",
        "end_time": "23:59:59"
        }
      ]
     }
}
```

system_hours.json

The `system_hours.json` file with the above configuration describes our rental service as being available every day of the week at any time for members and non-members (who do not have an account on the operator's application/site).
The test was carried out with a date of Sunday 10 July 2022. The expected result was the non-use of one of our operator's vehicles. However, Figure 6.11 shows that the calculated trip did use our shared vehicle.

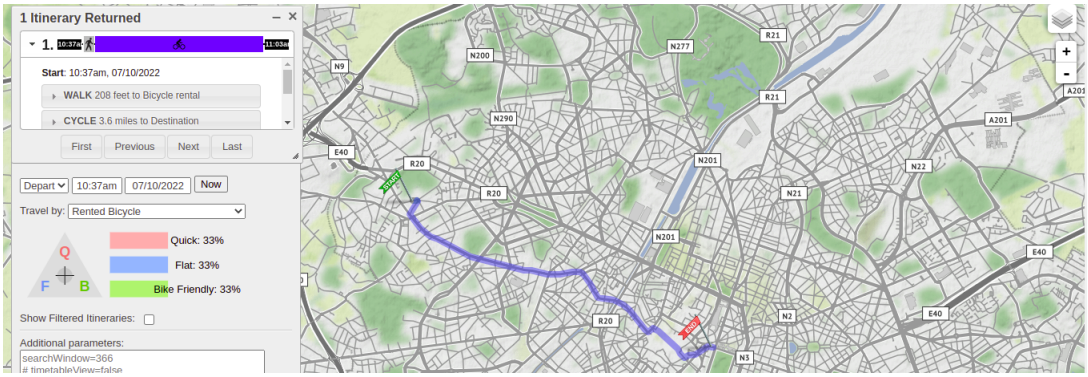


Figure 6.11: Trip using a vehicle from an operator which is not active at that time

Furthermore, if we change the hours when the service is available between 8am and 8pm, the result remains the same. We can therefore see that *OpenTripPlanner* does not seem to manage the `system_hours.json`.

Zones with restrictions

As a reminder, the `geofencing_zones.json` file allows you to define restrictions on certain zones. For example, this allows you to define a ban on dropping off a vehicle in a park or in a municipality.
In addition, bike-share providers are only present in certain cities. Therefore, it is not allowed to finish your trip in a city or region that is not part of the network.

In order to do this test, the file is configured as follows:

```
{
"last_updated": 1609866247,
"ttl": 86400,
"version": "2.2",
"data": {
  "type": "FeatureCollection",
  "features": [
    {
    "type": "Feature",
    "geometry": {
    "type": "MultiPolygon",
    "coordinates":
    [[[[4.3291112,50.8635934],[4.326547,50.8630313],[4.3230064,50.8644737],
    [4.3194445,50.8653743],[4.320689,50.8670671],[4.3213649,50.8667692],
    [4.3226471,50.8663054],[4.3240472,50.865977],[4.3259891,50.8656046],
    [4.3280597,50.8652592],[4.3291112,50.8635934]]]]
     },
     "properties": {
     "name": "Parc Elisabeth",
     "rules": [
      {
      "vehicle_type_id": ["2"],
      "ride_allowed": false,
      "ride_through_allowed": false
       }
```

```
        ]
      }
    }
  ]
 }
}
```

geofencing_zones.json

Figure 6.12 shows a visualisation on *QGis* of the area where we apply the restrictions.

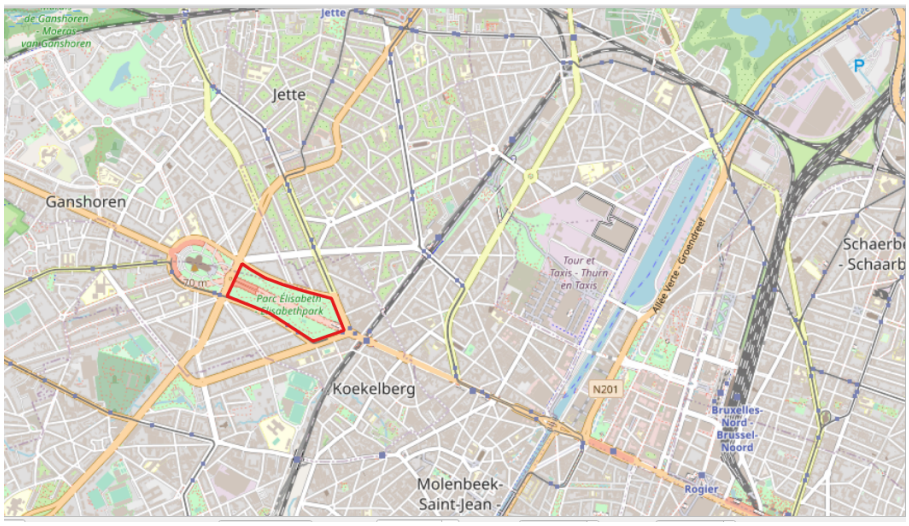


Figure 6.12: Area where restrictions are going to be applied

With the `ride_allowed` and `ride_through_allowed` parameters set to `false`, no vehicle can normally enter this area,yet this is not what we see (Figure 6.13).

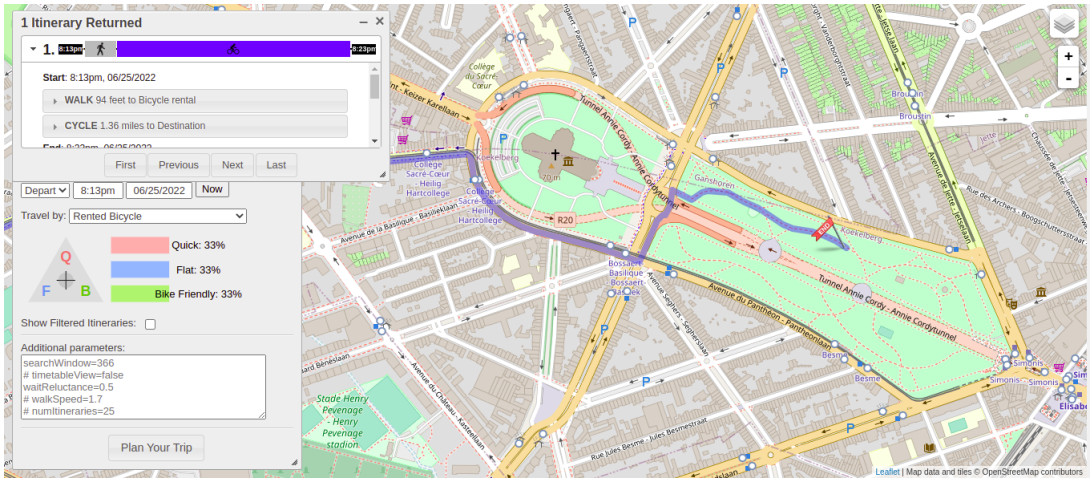


Figure 6.13: Vehicle entering an unauthorised area

Regardless of the combinations of `ride_allowed` and `ride_through_allowed` parameters, the results obtained are biased. As a result, it looks like that *OpenTripPlanner* does not take them into account.

6.1.4 Travelling salesman problem

Travelling salesman problem (TSP) can be an interesting optimisation problem to study from a mobility point of view. As a reminder, the travelling salesman problem consists of finding the shortest path to visit n area once and return to the starting point [24].
In order to solve this problem, the following question must be asked. In what order should the areas be visited in order to get the shortest tour?

Although simple to state, the problem is complex. In fact, it is even classified as *NP-hard* from the point of view of the mathematical field of combinatorial optimization [24].
Currently, *OpenTripPlanner* does not offer any concrete solutions for managing TSP. Indeed, a parameter close to it (`intermediatePlaces`) was available on OTP1 but the latter was content to visit the places in the requested order[6]. This parameter is currently unavailable in OTP2.[7]
Note that there is a solution to TSP in *PgRouting* [29].

[6]https://groups.google.com/g/opentripplanner-users/c/QtmCqshvPrU/m/IdyiJQH2AAAJ
[7]http://docs.opentripplanner.org/en/latest/OTP2-MigrationGuide/

Chapter 7

Experimental Integration

7.1 GBFS Integration by OpenTripPlanner

7.1.1 Description Algorithm and Case Study

As we showed earlier, the *GBFS* format, which can represent a set of vehicles from shared mobility, was not well supported by *OpenTripPlanner*. Our contribution will therefore focus on these important shortcomings, which make most of the calculated routes biased.
In order to do this, we will focus on some important files that are either partially supported by *OpenTripPlanner* or not supported at all. The idea is to bring to *OpenTripPlanner*, and consequently to open source, a complete management of the following files: `free_bike_status.json`, `station_status.json` as well as `geofencing_zones.json`.
More concretely, we will implement the following features:

- In the `free_bike_status.json` file we will prevent a user from taking a vehicle already reserved by someone else or already disabled.
- Also in the `free_bike_status.json` file, we will prevent the user from taking a vehicle that is not sufficiently loaded to make a trip, but rather direct the user to another vehicle that is close by and has a sufficient load.
- In the `station_status.json` file, we will prevent a user to start a trip from a station out of service, but rather direct it to a nearby and accessible station.
- Also in the `station_status.json` file, we will prevent a user to stop a trip to a station which is out of service, but rather direct it to a nearby and accessible ending station.
- In the `geofencing_zones.json` file, we will prevent a user to drop off a vehicle in an unauthorised area or even to cross an unauthorised area.

First, the *JSON* parser extracted the different data and created the `BikeRentalStation` objects containing the different vehicles. These stations include many fields defined in the *GBFS* files, namely a name, latitude, longitude, the number of available vehicles, the number of available docks,...

In the case of free-floating vehicles, it is the `free_bike_status.json` file that is parsed. Free-floating vehicles are also represented as stations. However, in order to mark their difference, the stations representing them have their `isFloatingBike` field set to `true` and the number of vehicles available is set `1`.
As the `free_bike_status.json` file is only partially managed, we need to add the `isReserved`

and `isDisabled` fields to our objects. If one of these 2 values is `true`, we consider the number of vehicles available at this "station" to be `0`. Thus, the vehicle will not be taken into account during the calculation of a journey.

Let's take the GBFS data we generated and try again:

- if the vehicle is disabled or already reserved, Figure 7.1 shows that this vehicle is no longer available.

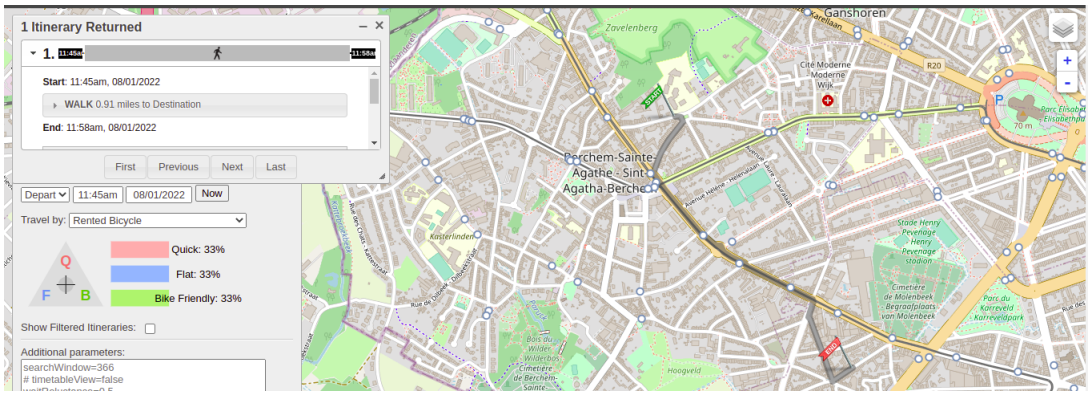


Figure 7.1: Trip returned when the only vehicle is disabled

- Conversely, Figure 7.2 shows the journey if the vehicle is available to the user.

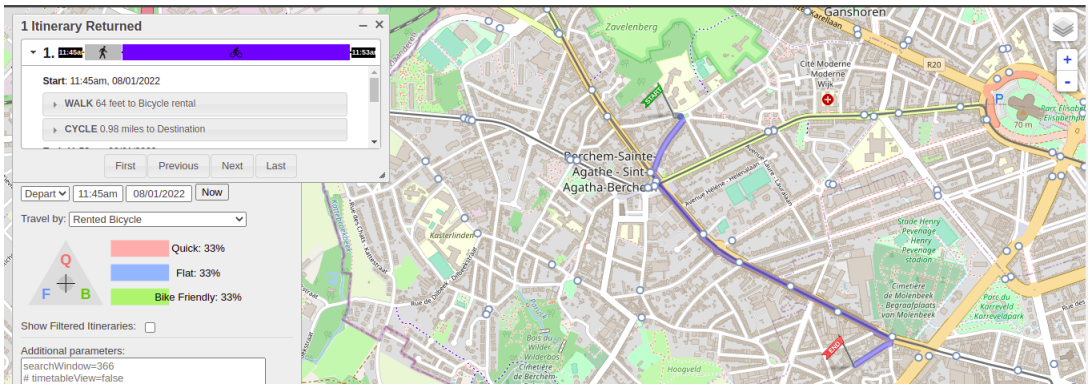


Figure 7.2: Trip computed with a shared vehicle

Secondly, we want a more complete management of the dock system. Again, the `station_status.json` file is parsed to create `BikeRentalStation` objects with the various parameters already mentioned.

We introduce the field `is_renting` and `is_returning`, if `is_renting` is `False`, then we set the value of the number of vehicles available at the station to `0`. If `is_returning` parameter is `False`, then we set the value of the number of free docks at the station to `0`.

Let's go back to our *GBFS* data and see if this time the data are properly managed:

- if there is no problem in our two stations, Figure 7.3 shows that our vehicle is used.

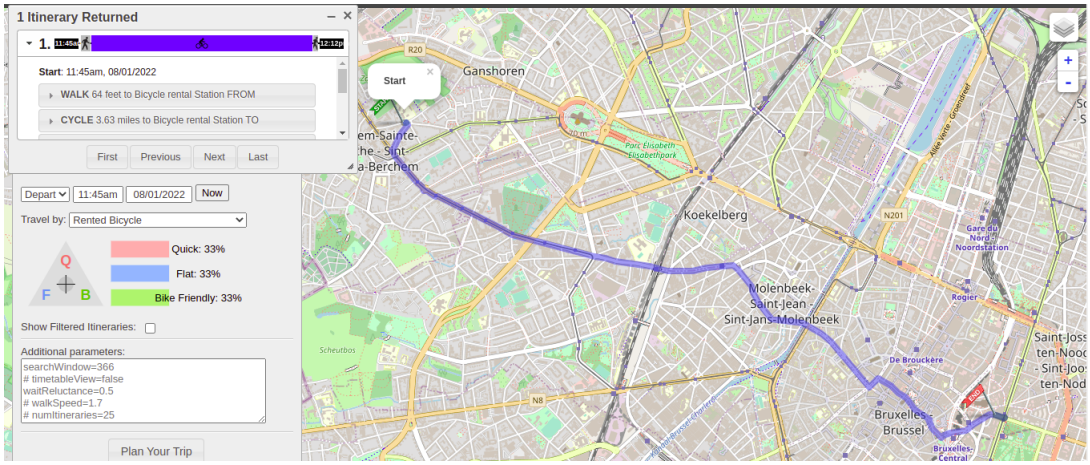


Figure 7.3: Trip computed when both stations are accessible

- Conversely, Figure 7.4 shows that the vehicle is no longer offered if one of the stations is out of service.

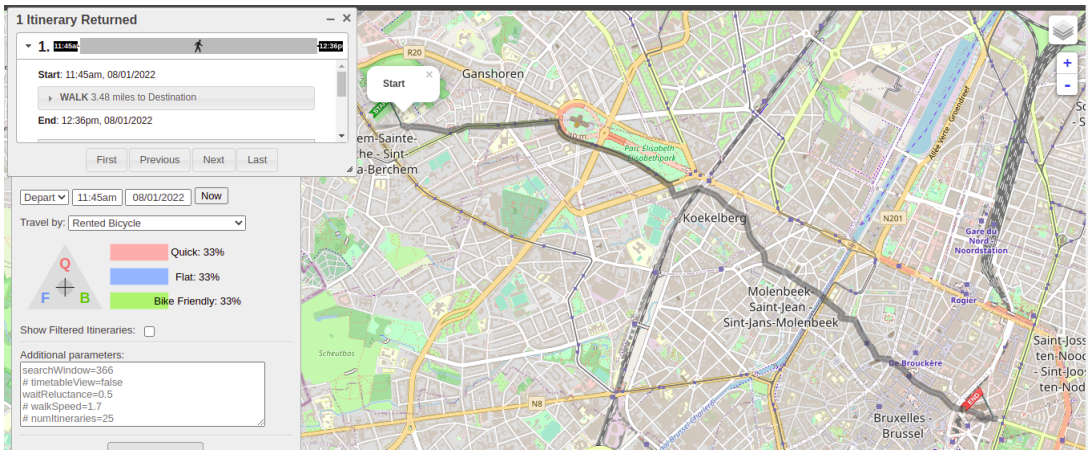


Figure 7.4: Trip computed when one of the station is out of service

We will now prevent a user from taking a vehicle that is not sufficiently loaded (or does not include enough fuel).
As a reminder, the itineraries are calculated using the A* algorithm (see section 2.1.4). We have kept the basics of the algorithm and made the necessary changes to support the `current_range_meters` parameter of the `free_bike_status.json` file.

Initially, we are at our starting node and we associate a `State` with it. A `State` is an object containing several parameters:

- The current time at this state.
- The distance travelled so far.
- The total weight so far (a weight is a sum of the distance and heuristic value between two nodes).
- A link to the previous state.
- The total distance travelled so far by a shared vehicle.
- If we currently use a shared vehicle or not.
- + other informations.

The state is the cornerstone of our algorithm. It is through it that we will be able to track the shared vehicles used and the distance travelled during the journey.
Once we have associated a state with our starting node, the A* algorithm will run normally until it arrives at a node whose associated `State` has its `isBikeRenting`[1] parameter set to `true`. This means that the algorithm switches transport modes and that the one currently used is a shared vehicle. The associated vertex contains the information of the station where the shared vehicle is located, and in our case of free-floating, the information of the vehicle itself. Note that once in a shared vehicle, the algorithm is designed to use it to a station or directly to destination.

From there, the execution of A* continues by looking at the adjacent vertices and calculates their `State` (which is an update from the current `State`). Thus it is possible to determine the number of metres travelled in a shared vehicle, the time once the vertex is reached, the total distance travelled, the total weight so far.
Thus, if the distance between two vertices (two `State`) is greater than our shared vehicle is capable of doing, we cancel the `State` (by not putting it in the queue of possible states for A* to visit). As a result, it is impossible for our shared vehicle to visit the vertex associated with that `State`. Note that if the `current_range_meters` parameter is not specified, we consider the vehicle to be human powered and therefore set the value of the parameter to `Double.MIN_VALUE`.

Now that the A* algorithm has been updated, let's resume our tests to check the correct handling of the `current_range_meters` parameter:

- If we put a low battery/little fuel (capacity of 1500 meters) and try to make a long journey, Figure 7.5 shows that the vehicle is not used anymore.

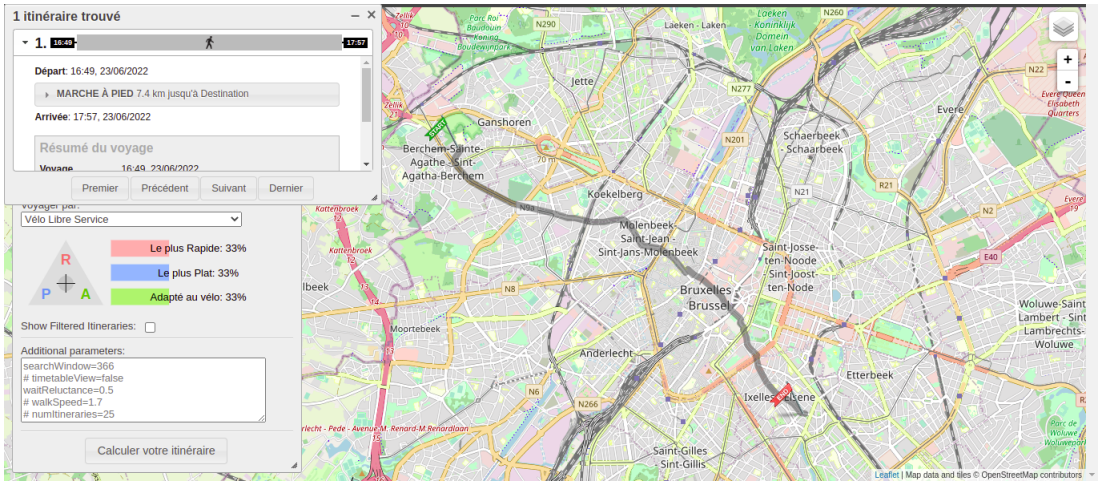


Figure 7.5: A too long trip computed with a shared vehicle

- If we put a better batery/more fuel (capacity of 15000 meters) and try to make the same journey, Figure 7.6 shows that the vehicle is accessible to make the journey.

[1] Actually bike renting is not appropriate because the sared vehicule could also be a car or a scooter

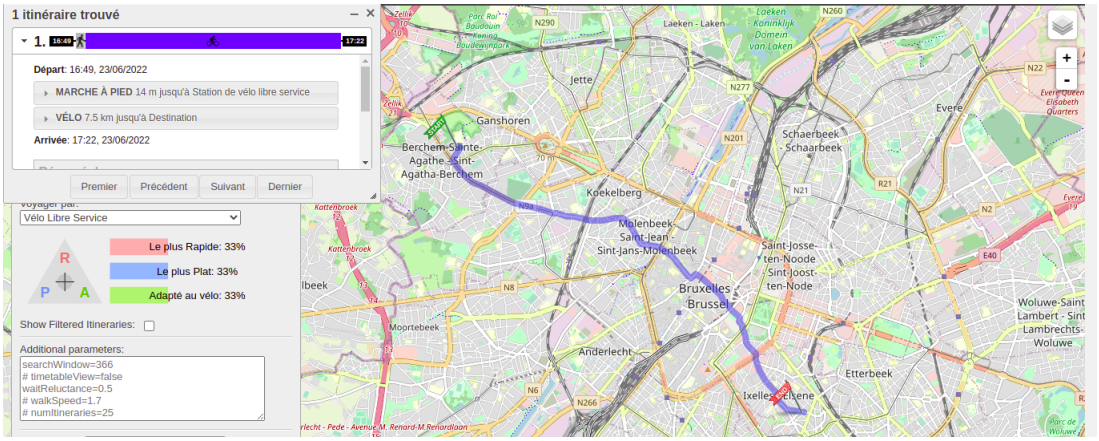


Figure 7.6: Long trip computed when a vehicle capable of doing so

- The combination with public transport is also affected, Figure 7.7 shows a test with a vehicle whose `current_range_meters` is at 500 whereas Figure 7.8 shows another test where the vehicle can travel up to 2500 meters.

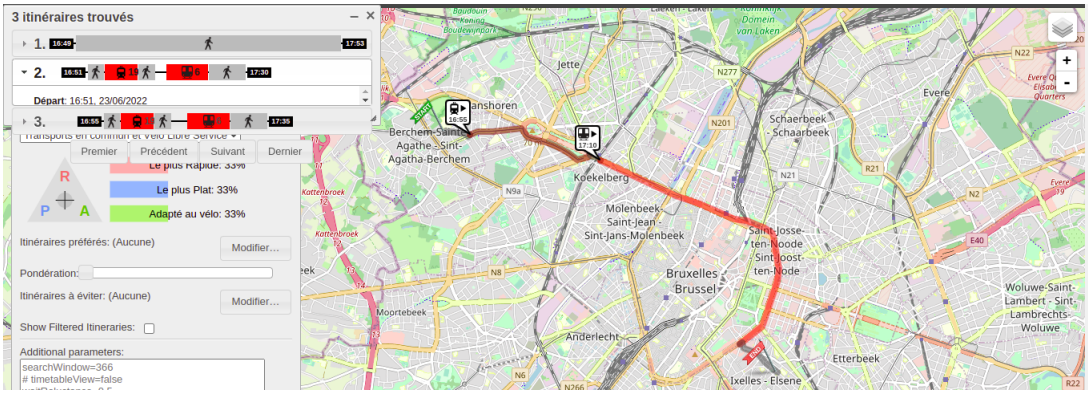


Figure 7.7: Trip computed when the vehicle can travel up to 500 meters

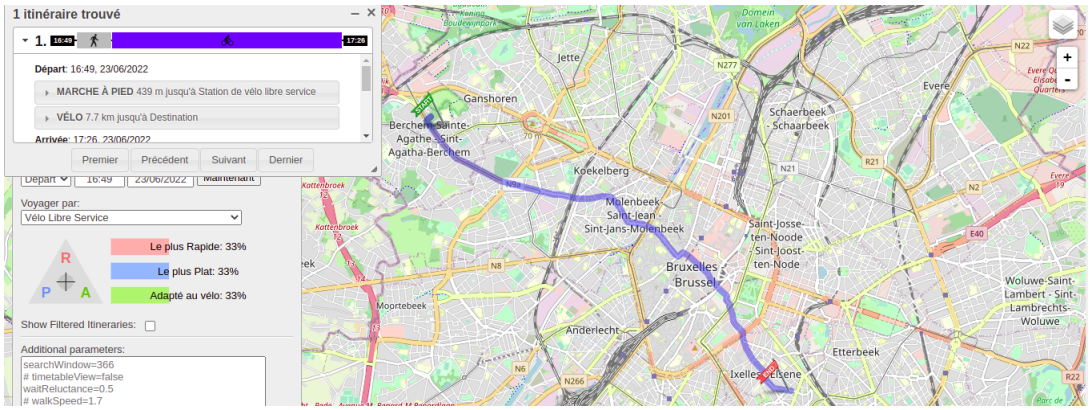


Figure 7.8: Trip computed when the vehicle can travel up to 2500 meters

The last part of our contribution consists in bringing a more complete management of the `geofencing_zones.json` file. To do this, each vehicle (`BikeRentalStation` object) should be associated with a new field representing a list of each area where a restriction exists and these restrictions (`GeofencingZones` objects).
Once A* is launched, it will run and visit the adjacent vertices, as soon as you are in a shared vehicle it must be checked at each visited vertex (`State` visited) if this vertex is in an area where restrictions exist (and are associated with our shared vehicle). If so, there are several cases to check:

- `ride_allowed` and `ride_through_allowed` are set to `true`. There is nothing special to do.
- `ride_allowed` is set to `true` and `ride_through_allowed` is set to `false`. In this case, we need to check whether our current vertex (current `State`) is inside the restriction area while the adjacent one is outside. If so, we cancel the next `State`.
- `ride_allowed` is set to `false` and `ride_through_allowed` is set to `true`. Here we need to check that if our destination is inside the restriction area, that our current vertex (current `State`) is outside the area and that the next vertex (next `State`) is inside the area. So we will modify the edge connecting the two vertices by changing its mode of travel (from shared vehicle to foot).
- `ride_allowed` and `ride_through_allowed` are set to `false`. We have to check if the next vertex and the destination are inside the area, then we modify the edge connecting our two vertices by changing its mode of travel (from shared vehicle to foot). If the next vertex is inside the area while the destination is not, then we cancel the next `State`.

Now that the A* algorithm has been updated, let's resume our tests to check the correct handling of the `geofencing_zones.json` file:

- With `ride_allowed` setting to `true` and `ride_through_allowed` setting to `false`, Figure 7.9 shows that the user can use the vehicle inside the the area while he must to find an other solution if the destination is outside the area, as shown in Figure 7.10.

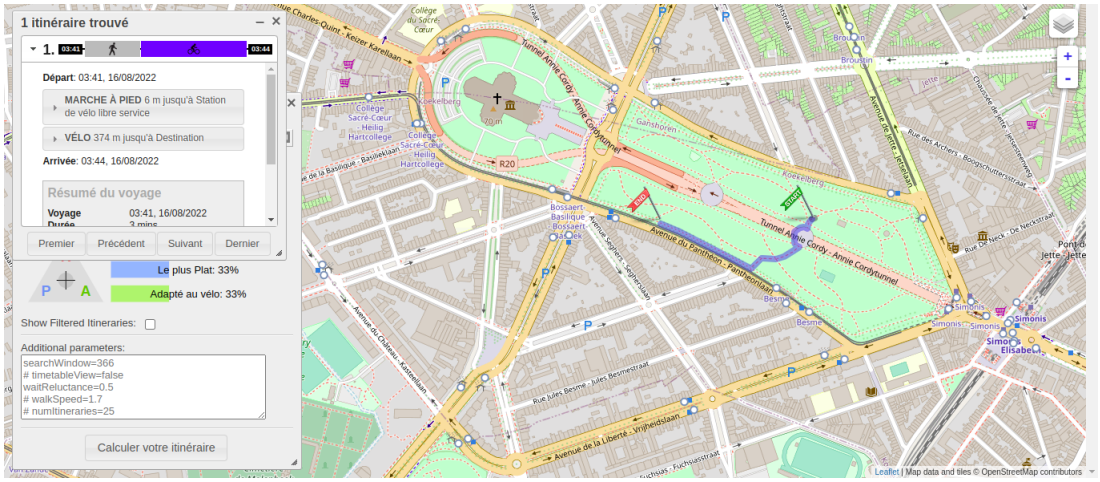


Figure 7.9: Trip computed when the area is accessible for the vehicle

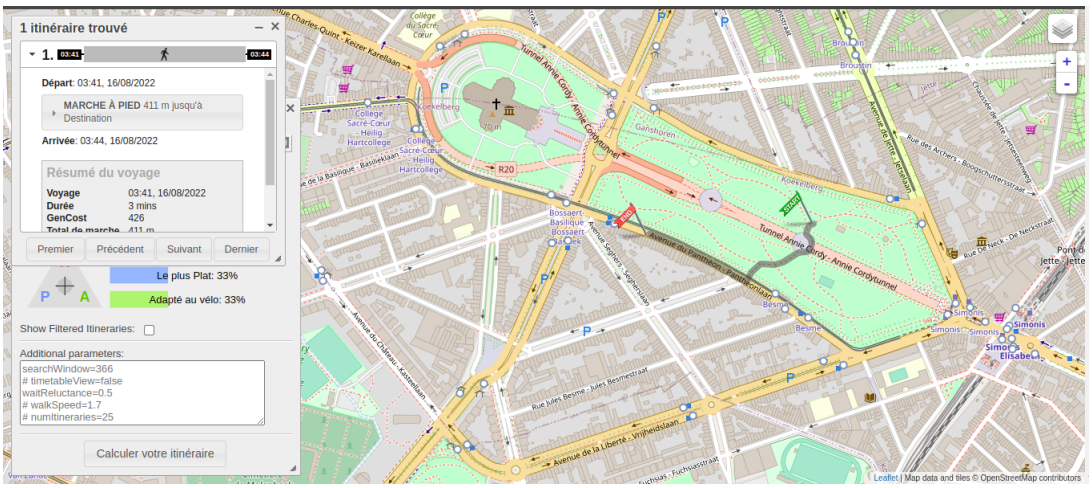


Figure 7.10: Trip computed when the area is unaccessible for the vehicle

- With `ride_allowed` setting to `false` and `ride_through_allowed` setting to `true` we could see that the vehicle can cross the road, represented by the Figure 7.11, while Figure 7.12 shows that the user must leave his vehicle just before entering the area.

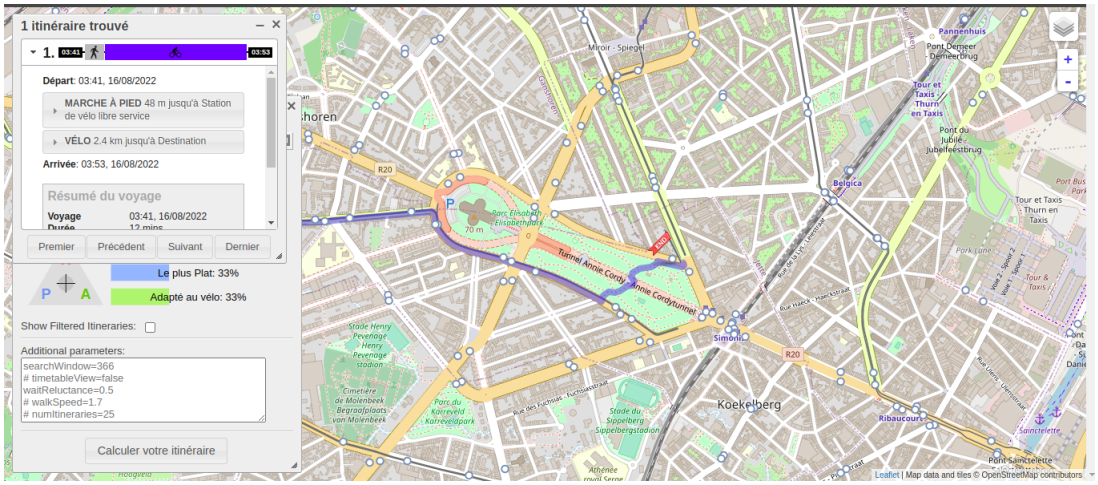


Figure 7.11: Trip computed when the vehicle can cross the area

Figure 7.12: Trip computed when the vehicle cannot cross the area

- `ride_allowed` setting to `false` and `ride_through_allowed` setting to `false` also produce expected and desired results.

7.1.2 Performances

After modifying *OpenTripPlanner* to better support *GBFS*, we will perform some performance tests to determine whether this support has a positive or negative impact on the calculation of a route for the user.

The various tests were carried out on a local *OpenTripPlanner* server. The machine used runs on Ubuntu 18.04 LTS, has 8GB RAM, an 512 Go SSD and an Intel Core i5-5300U CPU running at 2.30GHz (2 physical cores and 2 logical cores).

Our tests will be focus on the city of Brussels. In addition, we used syntethic *GBFS* data of more than 960 vehicles, each with different characteristics (note that Pony currently offers 964 vehicles in the city of Brussels).
We have also defined some regions as inaccessible, while others allow the user to complete his journey within them (`ride_allowed` parameter to `true`) but forbid the user to pass through these areas (`ride_through_allowed` parameter to `false`).

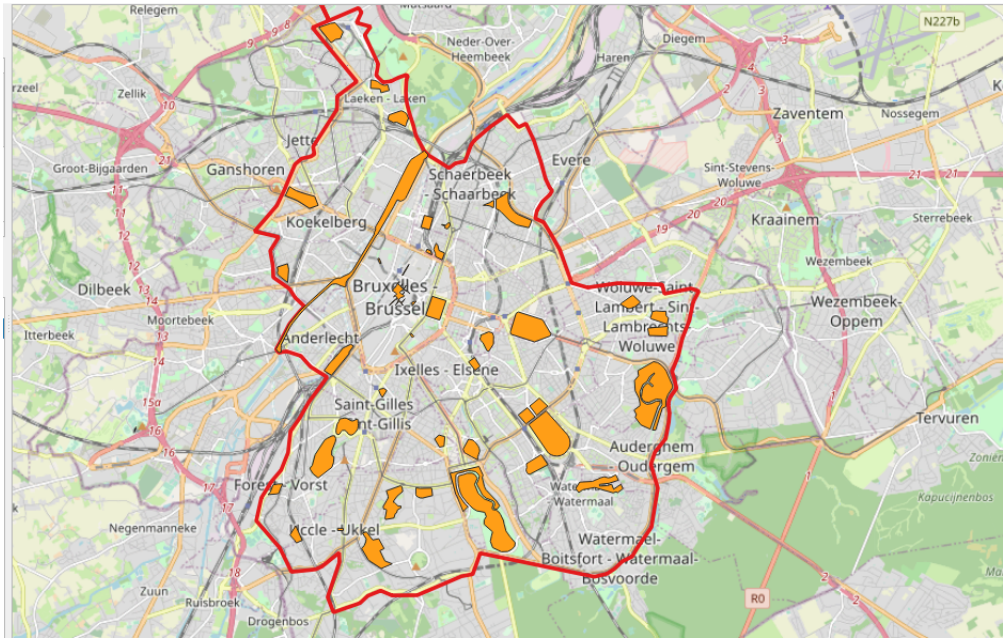


Figure 7.13: Map on *Qgis* showing restrictions for free float vehicles

- The red line around the city of Brussels represents the limitation. No vehicle can cross this line and therefore cannot leave the city.
- The yellow areas represent the areas where you can stop a vehicle but you cannot drive directly through these areas with a vehicle.

Without transit

Firstly we construct the graph, it contains |V| = 142740 as well as |E| = 380451. We have compared the performance of the different algorithms.

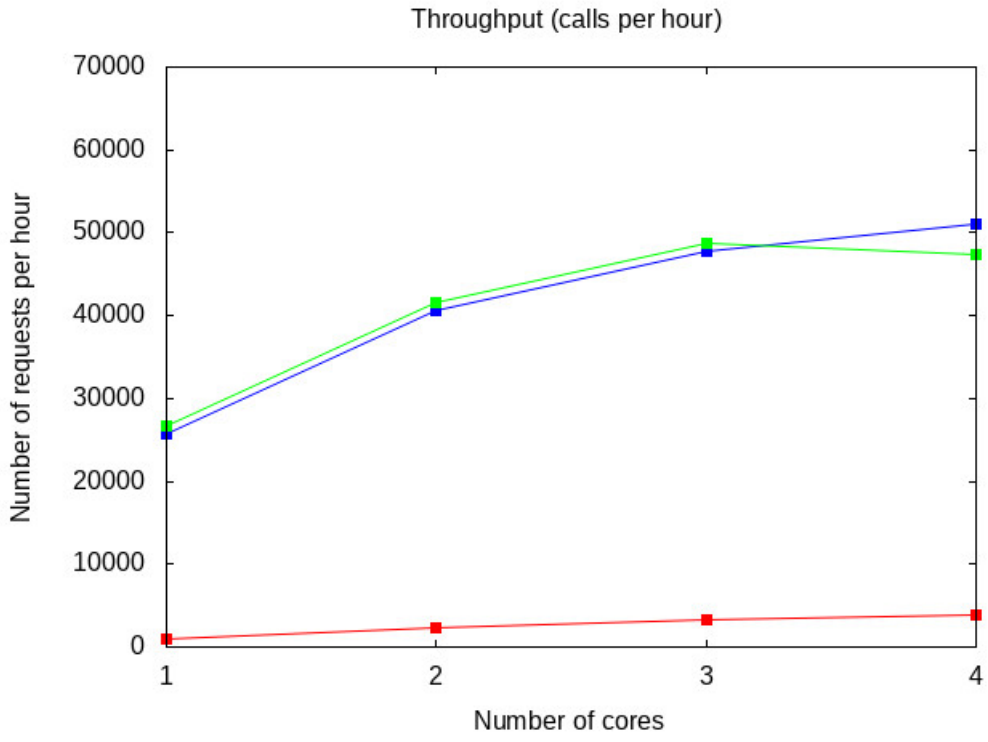


Figure 7.14: Performances without transit

- The blue line represents the original A* of *OpenTripPlanner*.

- The red line represents the A* supporting the `free_bike_status.json` file, containing in particular the parameter `current_range_meters`.
- The red line represents te A* supporting `free_bike_status.json` file as well as `geofencing_zones.json` file.

As we can see, thanks to multi-threading, A* can provide 50,000 responses per hour. However, this scaling gives poor performance for the "full A*" algorithm, where the number of executed requests drops to around 3000-4000 per hour.

With transit

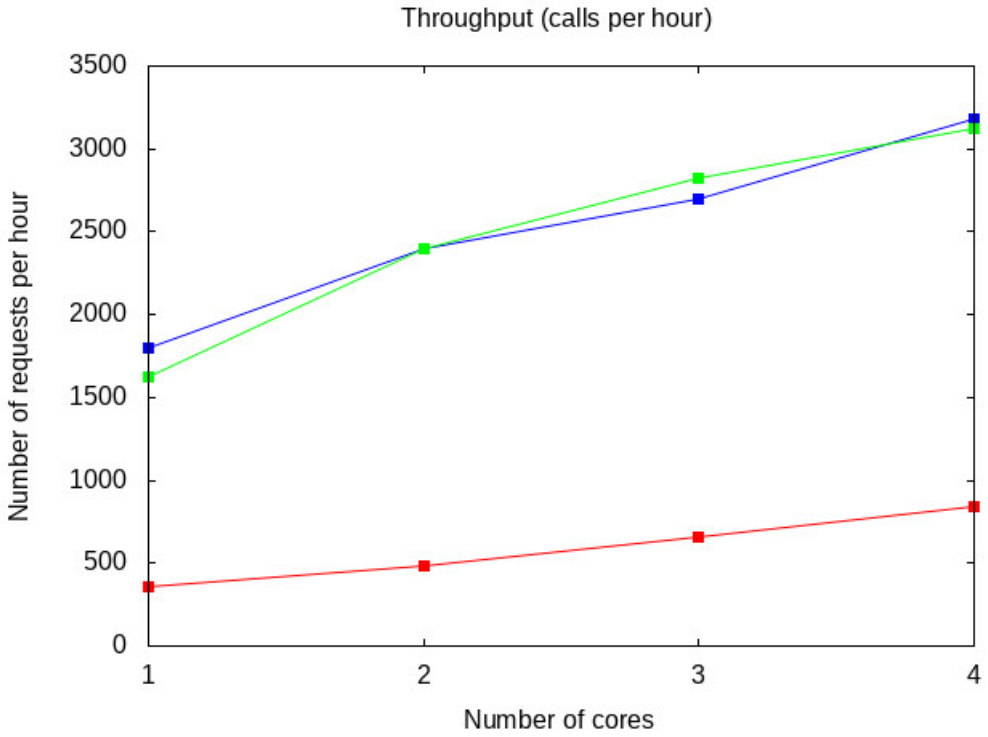


Figure 7.15: Performances with transit

We observe here also that the first 2 implemented versions of A* give more or less the same number of results per hour (around 3400) while the "full" version is below. They only execute more or less 800 queries per hour, i.e. 4 times less results.

7.2 OpenTripPlanner and MobilityDB

This section is about generating trajectories using *OpenTripPlanner* and connect *MobilityDB* on those trajectories. The vilualization will be done by *Qgis*.

7.2.1 Preparation

First of all, we create a database with the necessary extensions *Postgis*, *MobilityDB* as well as *Hstore*. In order to contact the *OpenTripPlanner* API, we set up a local web server with the OSM data from Brussels as well as the *GTFS* data regarding the transit network.

To display the trips, the *Qgis* tool is used. Instructions for downloading and installing it can

be found here[2]. In addition, a *Qgis* plugin called *Move*[3] is used. It allows to query *MobilityDB* databases and visualise objects moving over a given period of time

7.2.2 A Multi-modal trip

First, we generate a multi-modal trip for a person wanting to travel from a point A to a point B in Brussels.
After sending a request to the *OpenTripPlanner* REST API with the desired parameters such as:

- `date`
- `time`
- `mode` (represents transport modes to consider: `WALK`, `BIKE`, `TRANSIT`, `...`)
- `arriveBy` (specifies that the given time is when we plan to arrive)
- `maxWalkDistance` (specifies the maximum distance in meters that you are willing to walk)
- `....`

We get our trip and we convert it into a *MobilityDB* format which allows us to visualise temporal points moving over time. A visualisation of temporal points can be shown thanks to the Figure 7.16.

[2]https://www.qgis.org/fr/site/forusers/download.html
[3]https://github.com/mschoema/move

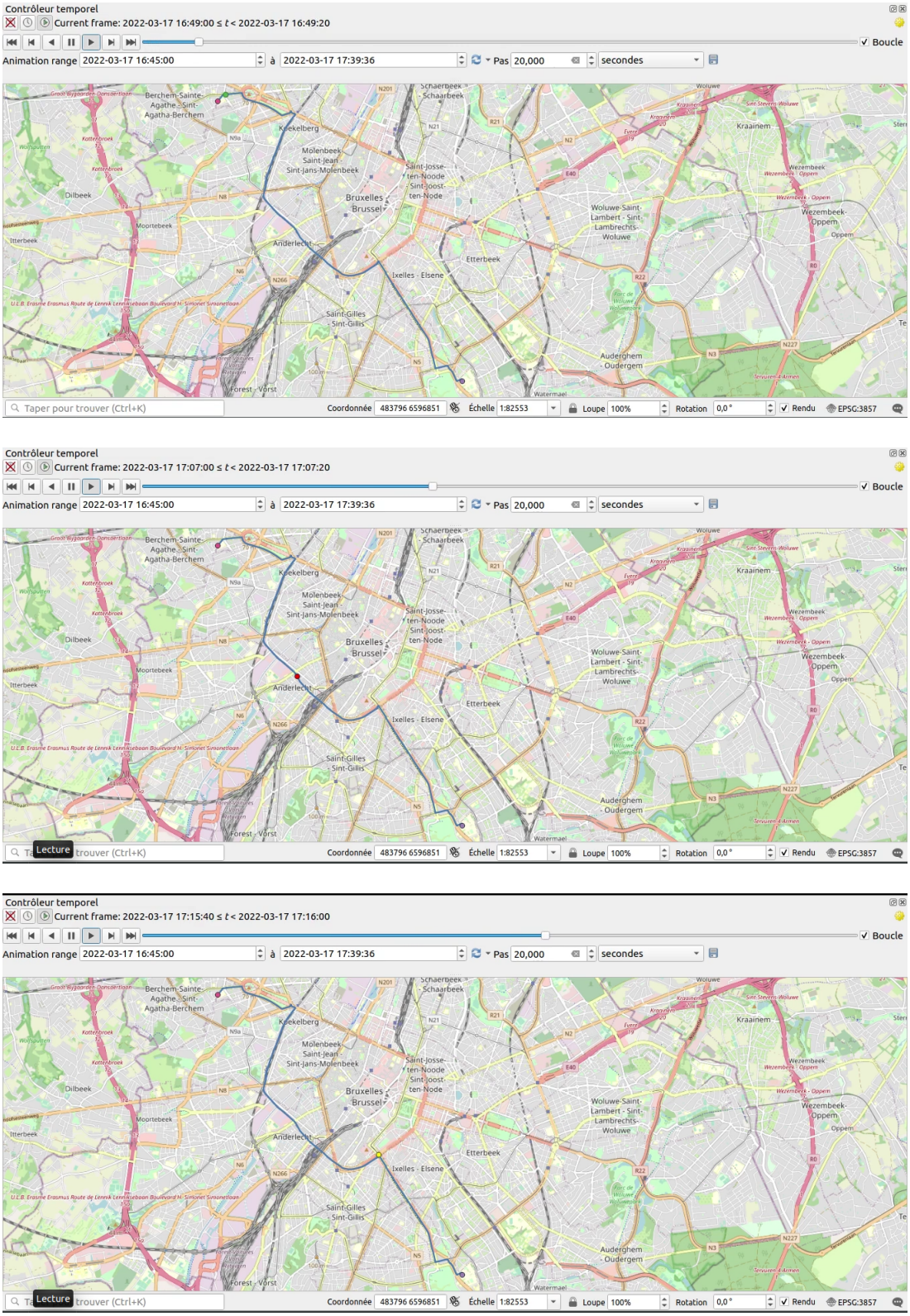


Figure 7.16: *MobilityDB points* representing people making a trip

Here you can see a person making a trip from point A to point B. The green dot represents the person walking. The yellow dot represents the person waiting. The red dot represents the person making the trip by public transport. The blue line represents the trip computed by *OpenTripPlanner*.

7.2.3 People Moving in Brussels

We could also generate multi-modal trips for a multitude of people.

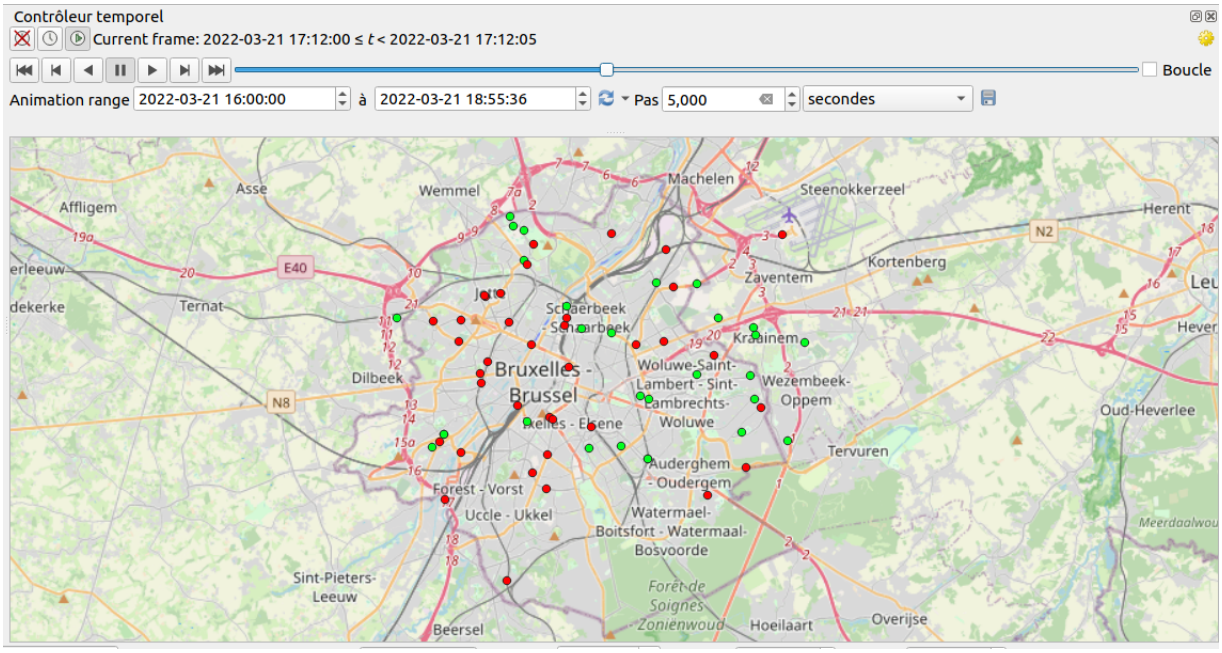


Figure 7.17: *MobilityDB* points representing people making trips

Again, each dot in Figure 7.17 represents a person having a trip (the colours of the dots being the same as in the previous example).

Chapter 8

Conclusion

8.1 Conclusion

As we have seen in this work, multi-modal routing requires a complete rethinking of the structure of our network compared to single-modal routing. Moreover, the returned paths are not necessarily the fastest or the shortest. But many other parameters come into play such as price, CO2 consumption, number of transfers, ...

All these changes require the design of adapted algorithms. As we have seen, Dijkstra is derived in several speed-up techniques which are then accelerated at the expense of a pre-processing phase. We can also note the implementation of algorithms which are not Dijkstra based and which focus essentially on transit routing, such as Raptor for example.
In addition, another area of reflection concerns data management, allowing us to partition our data in order to improve our performance (discussed briefly in the 3.3)

Many companies or open source projects are interested in multi-modal routing, among those reviewed, we have on the one hand *Google Maps* which is limited to bi-modal and where the user has no choice of preferences. On the other hand, we have *OpenTripPlanner* which is more customisable by the user at the expense of simplicity of use and which has no app available on Android or IOS.

We also saw the need for different data formats and their support. There is a desire to standardise these formats to facilitate their integration. These formats continue to evolve today, integrating more and more data. It is therefore necessary to monitor these developments in order to integrate them into the various existing tools.
Not only does research need to continue in order to develop these tools, but the various market players should also be centralised. Although multi-modal trips are offered to the user, he has to install the applications of the different services in order to use their vehicles, which does not facilitate the adoption of multi-modality.
However, there is no doubt that solutions will be found in the upcoming months/years thanks to the political pushes to implement multi-modality within cities at the expense of the private car. To take the small city of Brussels as an example, The Brussels Intercommunal Transport Company, the *STIB/MIVB* is currently working on a multi-modal application called MoveBrussels[1].

[1] https://maasification.com/applications/by-application/movebrussels/

Bibliography

[1] Netex extension for new modes (https://www.netex-cen.eu/wp-content/uploads/2021/03/netex-extension-for-new-modes-detailed-scope-v04.pdf). 2020.

[2] Aaron Antrim, Sean J Barbeau, et al. The many uses of gtfs data–opening the door to transit and multimodal applications. *Location-Aware Information Systems Laboratory at the University of South Florida*, 4, 2013.

[3] Fabrizio Arneodo. Public transport network timetable exchange (netex) : Introduction (https://www.netex-cen.eu/wp-content/uploads/2015/12/01.netex-introduction-whitepaper_1.03.pdf). 2015.

[4] Hannah Bast. Car or public transport—two worlds. In *Efficient Algorithms*, pages 355–367. Springer, 2009.

[5] Hannah Bast, Erik Carlsson, Arno Eigenwillig, Robert Geisberger, Chris Harrelson, Veselin Raychev, and Fabien Viger. Fast routing in very large public transportation networks using transfer patterns. In *European Symposium on Algorithms*, pages 290–301. Springer, 2010.

[6] Hannah Bast, Daniel Delling, Andrew Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F Werneck. Route planning in transportation networks. In *Algorithm engineering*, pages 19–80. Springer, 2016.

[7] Hannah Bast, Matthias Hertel, and Sabine Storandt. Scalable transfer patterns. In *2016 Proceedings of the Eighteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 15–29. SIAM, 2016.

[8] Annabell Berger, Daniel Delling, Andreas Gebhardt, and Matthias Müller-Hannemann. Accelerating time-dependent multi-criteria timetable information is harder than expected. In *9th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'09)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2009.

[9] Altannar Chinchuluun and Panos M Pardalos. A survey of recent developments in multiobjective optimization. *Annals of Operations Research*, 154(1):29–50, 2007.

[10] Thomas Craig and Weston Shippy. Gtfs flex–what is it and how is it used? 2020.

[11] D Delling. Engineering and augmenting route planning algorithms (ph. d. thesis). *University of Karlsruhe*, 2009.

[12] Daniel Delling, Julian Dibbelt, Thomas Pajor, and Tobias Zündorf. Faster transit routing by hyper partitioning. In *17th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[13] Daniel Delling, Andrew V Goldberg, Thomas Pajor, and Renato F Werneck. Customizable route planning in road networks. *Transportation Science*, 51(2):566–591, 2017.

[14] Daniel Delling, Thomas Pajor, and Dorothea Wagner. Accelerating multi-modal route planning by access-nodes. In *European Symposium on Algorithms*, pages 587–598. Springer, 2009.

[15] Daniel Delling, Thomas Pajor, and Renato F Werneck. Round-based public transit routing. *Transportation Science*, 49(3):591–604, 2015.

[16] Edsger W Dijkstra et al. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[17] Yann Disser, Matthias Müller-Hannemann, and Mathias Schnee. Multi-criteria shortest paths in time-dependent train networks. In *International Workshop on Experimental and Efficient Algorithms*, pages 347–361. Springer, 2008.

[18] S Kiavash Fayyaz S, Xiaoyue Cathy Liu, and Guohui Zhang. An efficient general transit feed specification (gtfs) enabled algorithm for dynamic transit accessibility analysis. *PloS one*, 12(10):e0185333, 2017.

[19] Lennart Frede, Matthias Müller-Hannemann, and Mathias Schnee. Efficient on-trip timetable information in the presence of delays. In *8th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'08)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2008.

[20] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *International Workshop on Experimental and Efficient Algorithms*, pages 319–333. Springer, 2008.

[21] Andrew V Goldberg and Chris Harrelson. Computing the shortest path: A search meets graph theory. In *SODA*, volume 5, pages 156–165. Citeseer, 2005.

[22] Victor Goossens. Exploring the state of the art of multi-modal routing. 2009.

[23] Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[24] Karla L Hoffman, Manfred Padberg, Giovanni Rinaldi, et al. Traveling salesman problem. *Encyclopedia of operations research and management science*, 1:1573–1578, 2013.

[25] Dominik Kirchler. *Efficient routing on multi-modal transportation networks*. PhD thesis, Ecole Polytechnique X, 2013.

[26] Mohammad Sultan Mahmud, Joshua Zhexue Huang, Salman Salloum, Tamer Z Emara, and Kuanishbay Sadatdiynov. A survey of data partitioning and sampling methods to support big data analysis. *Big Data Mining and Analytics*, 3(2):85–101, 2020.

[27] Bibiana McHugh. Pioneering open data standards: The gtfs story. *Beyond transparency: open data and the future of civic innovation*, pages 125–135, 2013.

[28] Matthias Müller-Hannemann and Mathias Schnee. Finding all attractive train connections by multi-criteria pareto search. In *Algorithmic Methods for Railway Optimization*, pages 246–263. Springer, 2007.

[29] Florian Nadler. Traveling salesman problem in pgrouting : https://www.cybertec-postgresql.com/en/traveling-salesman-problem-with-postgis-and-pgrouting/. 2021.

[30] Alberto Maria Segre. On combining multiple speedup techniques. In *Machine Learning Proceedings 1992*, pages 400–405. Elsevier, 1992.

[31] Stephan Seufert, Avishek Anand, Srikanta Bedathur, and Gerhard Weikum. Ferrari: Flexible and efficient reachability range assignment for graph indexing. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 1009–1020. IEEE, 2013.

[32] Lenie Sint and Dennis de Champeaux. An improved bidirectional heuristic search algorithm. *J. ACM*, 24(2):177–191, apr 1977.

[33] Bezaye Tesfaye, Nikolaus Augsten, Mateusz Pawlik, Michael H. Böhlen, and Christian S. Jensen. Speeding up reachability queries in public transport networks using graph partitioning. *Inf. Syst. Frontiers*, 24(1):11–29, 2022.

[34] Chi Tung Tung and Kim Lin Chew. A multicriteria pareto-optimal path algorithm. *European Journal of Operational Research*, 62(2):203–209, 1992.

[35] Sibo Wang, Wenqing Lin, Yi Yang, Xiaokui Xiao, and Shuigeng Zhou. Efficient route planning on public transportation networks: A labelling approach. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 967–982, 2015.

[36] Huanhuan Wu, Yuzhen Huang, James Cheng, Jinfeng Li, and Yiping Ke. Reachability and time-based path queries in temporal graphs. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, pages 145–156. IEEE, 2016.

[37] Esteban Zimányi. Berlinmod benchmark on mobilitydb https://github.com/mobilitydb/mobilitydb-berlinmod. 2020.

[38] Esteban Zimányi, Mahmoud Sakr, and Arthur Lesuisse. MobilityDB: A mobility database based on PostgreSQL and PostGIS. *ACM Trans. Database Syst.*, 45(4), December 2020.