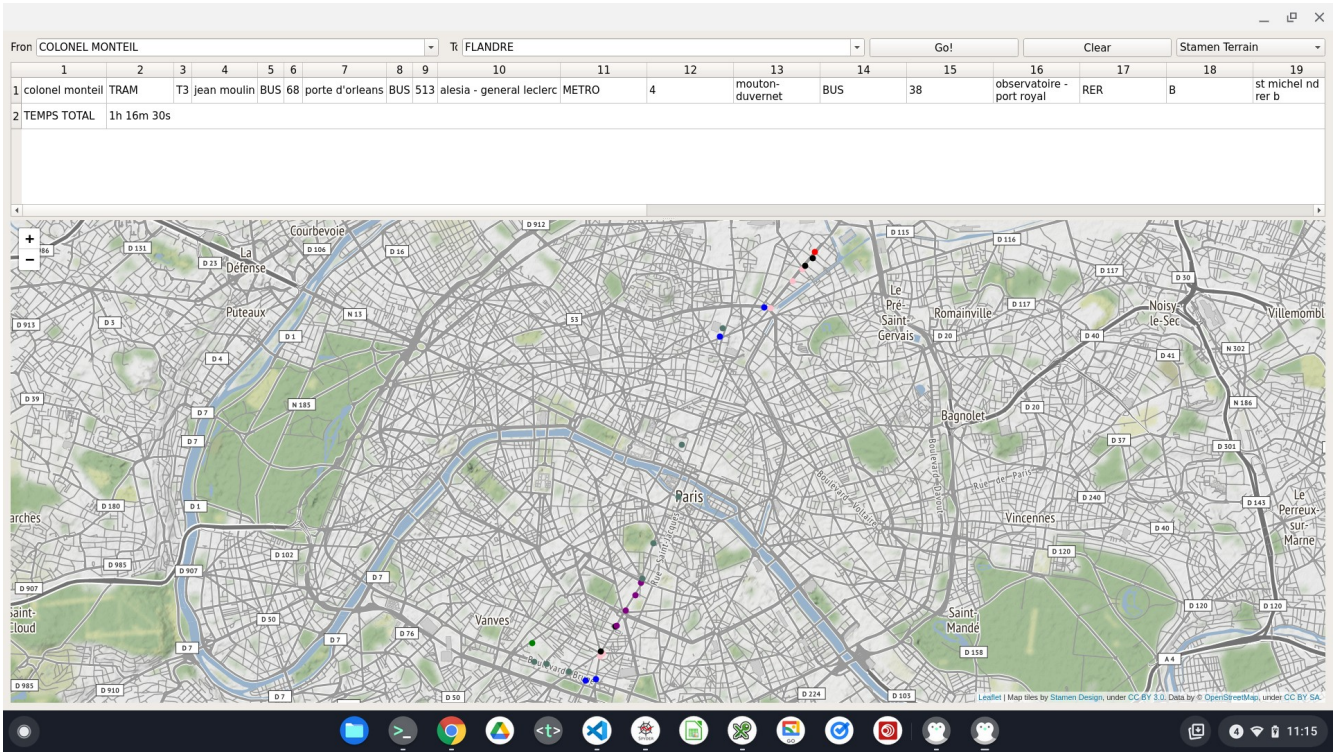
Paris City Mapper

Chorouk Mouss · Moussa Doumbiya · Gabin Fay



We developped a simple city mapper that can

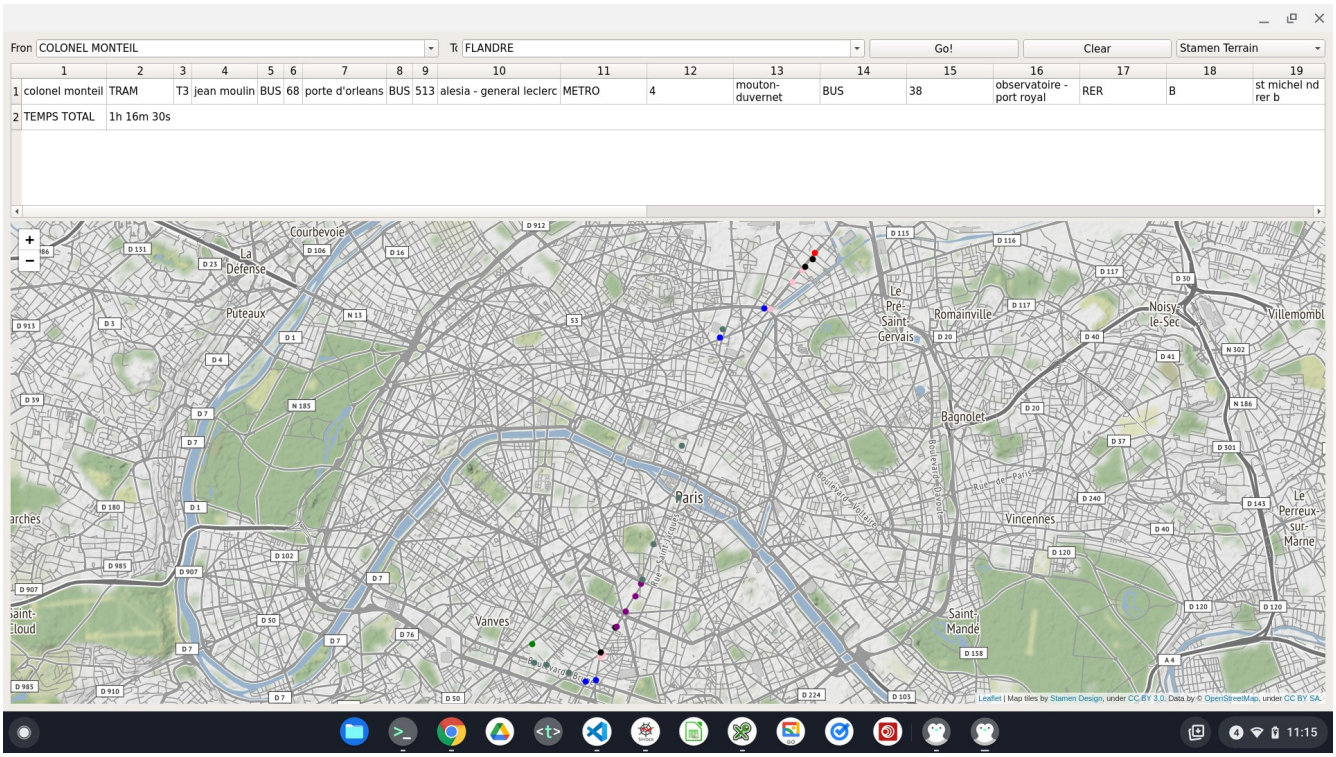
1. Compute shortest path between two public transport stations.
2. Prevent you from walking under the rain
3. Display the official RATP plan of every route

Github :

Docker :

TABLE OF CONTENTS

O.Tutorial & Installation



Select by typing or by clicking on the map two points.

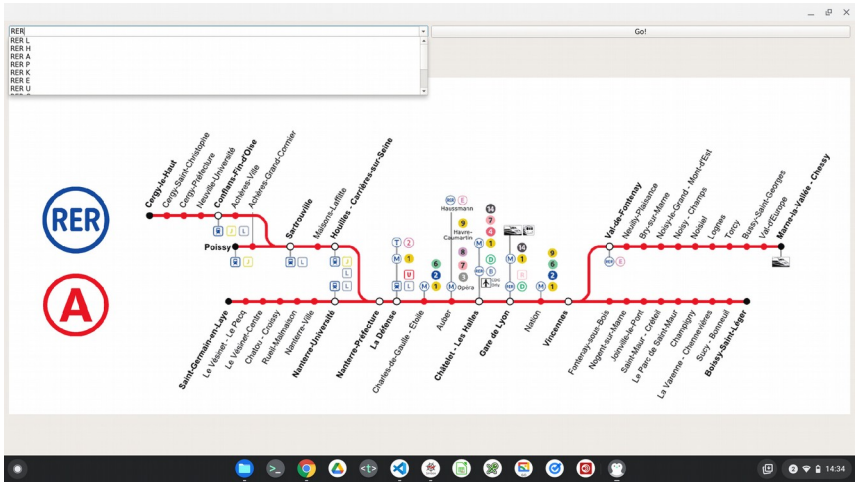
Press the button ‘Go’

Wait for it to compute, it outputs the shortest path in the table and with dots on the map, along with its duration

Rest assured you won’t walk under no rain

Displaying plans requires to launch a separate app

You choose the route you want to display and it displays the plan



Installation

We used docker to containerized our app.

The image is stored in a Docker Hub repository :

https://hub.docker.com/repository/docker/gabingabin/projet_sql_paris13

Running instructions

@localhost$ xhost +"local:docker@" === to allow the forwarded docker GUI to be displayed

@localhost$ docker pull -a gabingabin/projet_sql_paris13

City mapper:

@localhost$ docker run -it -e DISPLAY=unix$DISPLAY -v /tmp/.X11-unix:/tmp/.X11-unix -u gabin --shm-size=500m projet_sql_paris13 python /home/gabin/thegraphway.py

Plan:

@localhost$ docker run -it -e DISPLAY=unix$DISPLAY -v /tmp/.X11-unix:/tmp/.X11-unix -u gabin --shm-size=500m projet_sql_paris13 python /home/gabin/plan.py

Weather:

@localhost$ docker run -it -e DISPLAY=unix$DISPLAY -v /tmp/.X11-unix:/tmp/.X11-unix -u gabin --shm-size=500m projet_sql_paris13 python /home/gabin/plan.py

Opening a shell in the container:

@localhost$ docker run -it -e DISPLAY=unix$DISPLAY -v /tmp/.X11-unix:/tmp/.X11-unix -u gabin --shm-size=500m projet-sql-paris13 bash

gabin@container$ cd *home*/gabin

gabin@container$ vim whichever file you'd want to inspect and run the experiments (clustering.py, parse.py)

Troubleshooting

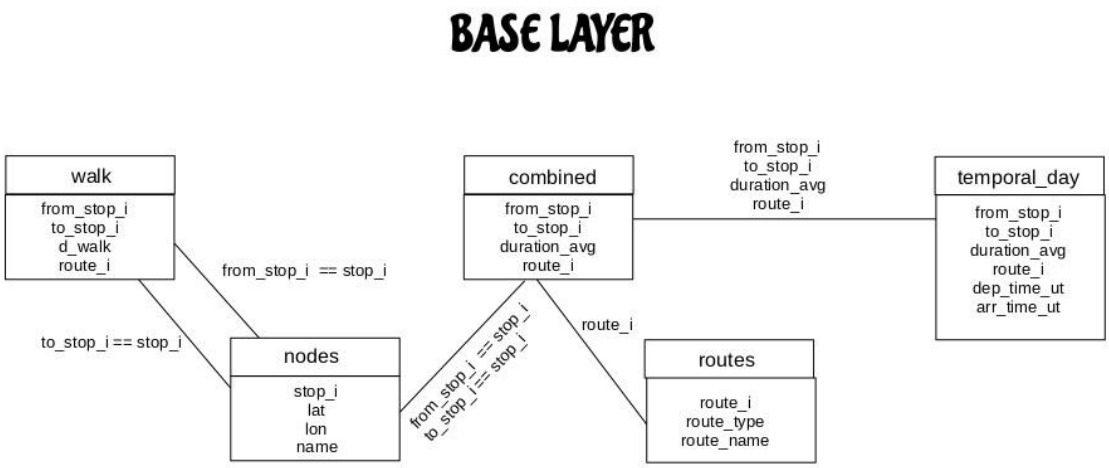
@localhost$ sudo apt-get install '^libxcb.*-dev' === in case you get a dependency issue (QT's dependencie ain't properly installed with pip)

@localhost$ ln -s /usr/lib/x86_64-linux-gnu/libxcb-util.so.0.0.0 /usr/lib/x86_64-linux-gnu/libxcb-util.so. ==== if libxcb-util.so.1 is missing

I. Tech stack

I.1. City Mapper

File : parse.py to parse the data thegraphway.py to run the app



nodes : relative to stops. Links each nodes name, latitude and longitude with its identifier

routes : defines public transport route that compose the network

combined : links nodes together by routes. there’s a line if there’s an elementary trip between two nodes. Hence, combined naturally defines edges of the public transport graph.

walk : same role as combined but for the walk. Links nodes together if they are at a distance < to 1km.

temporal_day : This defines the departure_time of each trip defined in combined

Computation

All computations are done with python. We use pandas DataFrame to work with data in python.

We use postgresql as database system and we use psycopg2 to interact with it within python.

We use sqlalchemy to read postgresql tables into pandas DataFrames.

We tested the program on a local database, but the actual program who runs in a docker container uses a remote database hosted on a VPS.

We read the SQL schema within python.

We first read the csv as pandas DataFrame, and then turn those into postgresql tables with an utility function.

Shortest path algorithm

We use networkx to work with graphs.

We read combined and walk as DataFrame, concatenate the two of them to allow both walking and public transport routes path in the shortest path, and turn this array into a weighted graph with average duration as weight. We got average duration from walk dividing the distance by 2.5 m.s^-1 which is a reasonable speed walking speed for a human.

The user chooses a departure and arrival stop. The program extract the identifiers (from_stop_i, to_stop_i) of those stops. Those are two vertices in our graph.

We compute Djikstra shortest path algorithm and it outputs vertices of the shortest path.

From those stops, we get detailed info on the path taken (all the routes, etc)

Adding Waiting Time

The algorithm doesn’t take into account the waiting time. As it is a difficult issue (shortest path in time windows), we assume it is the optimal path (false assumption) and compute the waiting time afterwards.

For this, we perform SQL queries on temporal_day to get the next trip available every time we need to take a new route.

The actual duration considering waiting times might be doubled the duration without taking them into account. We acknowledge its pretty

bad. It takes too much buses, the path ain’t in Google Maps best path, the path is composed of too many different routes.

UI

Every route in the path is displayed in the table along with the time (+ waiting time)

Nodes composing the path are displayed as coloured dots on the map depending on the route we are in when we cross those nodes.

If you double click on the table, it adds segments linking those nodes, but for a weird reason, folium (the python module that emulates JS Leaflet) doesn’t print them with the right colour

I.2 – Weather

File : weather.py

As students, we got free access to the developer tier of the OpenWeatherMap API, allowing us to compute up the weather for up to 3000 locations per minute.

Even after clustering the data, we still had ~ 8000 supernodes => The program would take 3 minutes to run.

Therefore, we cut Paris into 160 evenly spaced cells, associated to each node its cell number, computed the weather for each cell, removed from walk nodes where it is raining, and used this modified walk table as the new walk in the previous programs.

It is pretty cool, but when it rains in a cell, it also rains in almost all the other cells. So we might juste have got a single node weather and decided to keep or throw walk accordingly.

I.3 – Plans

File : plan.py

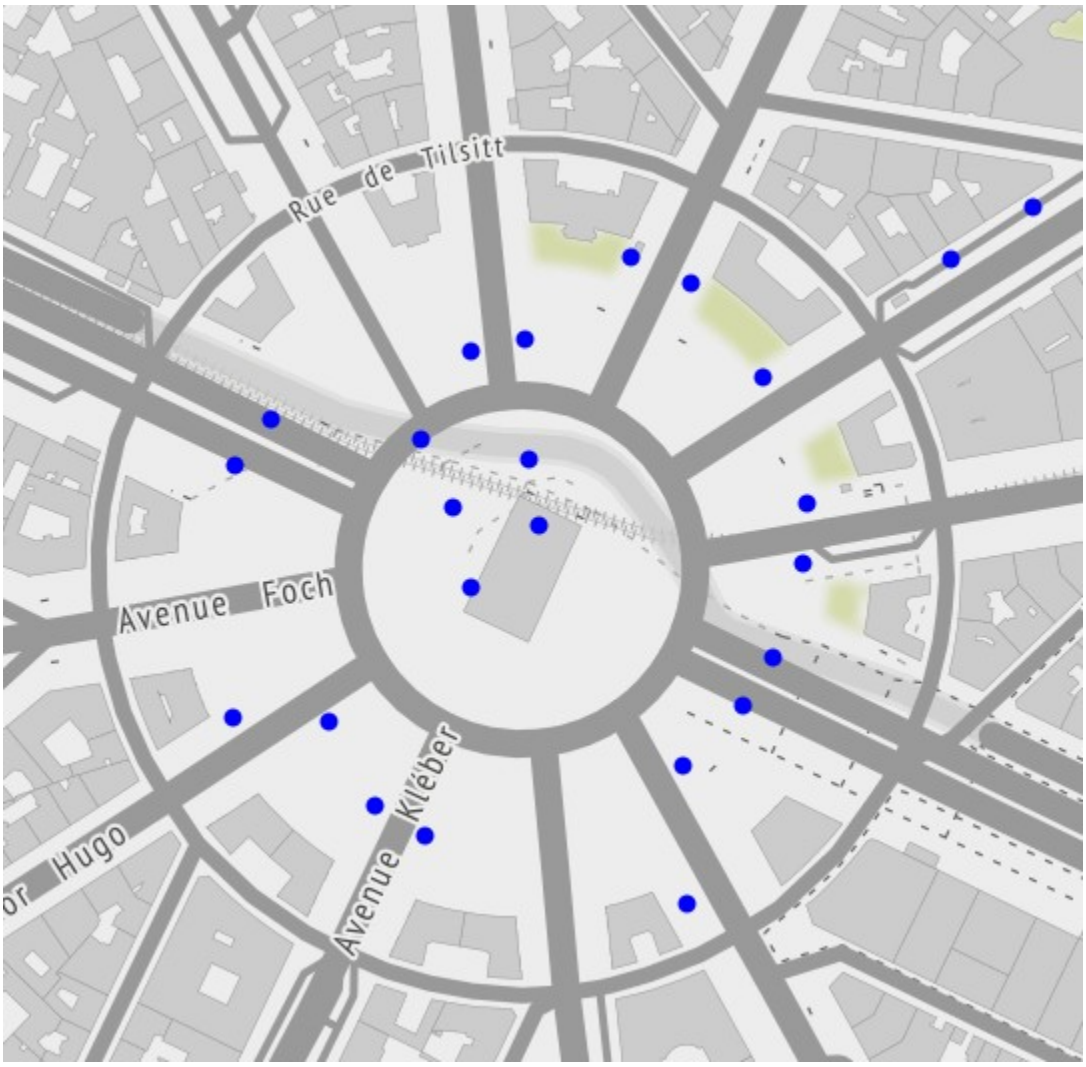
The RATP doesn’t have an easy to use API and we can’t scrape immediately its website.

We noticed the plan is the first image displayed by Google Image when you query the route name + ‘Paris’.

So, we used a python scraping program that scrapes images directly from Google Images. It allows to download up to 100 images for each query and you can easily specify the image type you want. We only needed to get the first image of each query.

II – EXPERIMENTS

II.1 – Clustering :



File : clustering.py to parse the data smallest.py for the app

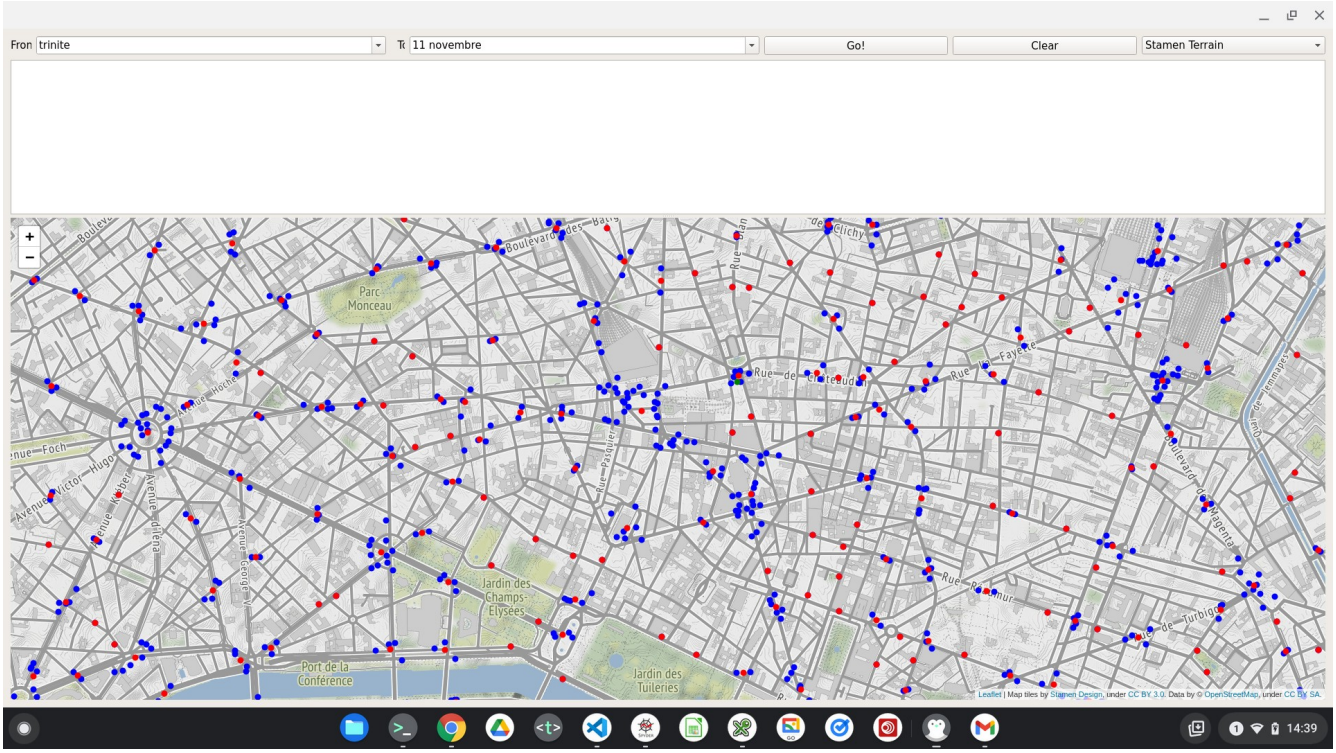
The previously computed shortest path ain’t optimal because it is composed of too many different routes, leading to increased waiting time.

We found interesting to find the path with the smaller number of nodes == compute the shortest path in the unweighed graph. But, the result was 100% walk as biggest walk elementary trips are 1km == 10 minutes. One simple issue was to kill all walk trips > 200m. But, before we figured this out, we thought walk needed not to be used at all.

But, several nodes corresponds to the same physical spot but are represent with different names and stop_i if they are served by different routes. Each route creates its own stops with it. So, if walk was removed, we needed a way to connect those close nodes together.

We decided to merge those nodes into one, called a supernode, and to replace all stop_i in combined by its corresponding supernode_i.

We used the DBSCAN algorithm to compute clusters. It allows the existence of single-nodes clusters and to cluster without knowing the number of clusters you want.



Along with this approach is the creation of four new SQL tables:

Cluster : associate to each node its cluster label. Label = -1 indicates a single-node cluster.

Supernodes : we build a table supernodes to link each node to its supernode (the supernode is chosen has the node with the smallest name)

Combxsuper : the modified combined table : each stop_i is replaced by its superstop_i

Walkxsuper : the modified walk table : each stop_i is replaced by its superstop_i

We then tried our idea (in smallest.py). Comb is replaced by combxsuper, and walk by walkxsuper.

In the end, the idea wasn’t fruitful, the trip after computing waiting time is longer than Google Images one even though it is composed of less nodes.

II.2. THE TP5 WAY

File : thetp5way.py

Here, we flattened the table combined to link each stop to routes that pass through it

We then used this table to build a new tablecalled stoproutename to replicate the algorithm of the tp5, where you can select the max number of hops.

Again, the issue is that there are 10 stops corresponding to chatelet for example and they aren’t linked in combined, so you need to merge them into one for the algorithm to work.

```
projet=# select * from routes where route_name ='A';
 route_type | route_name | route_i
------------+------------+---------
          2 | A          |     164
          2 | A          |     165
          2 | A          |     166
          2 | A          |     167
          2 | A          |     168
          2 | A          |     169
          2 | A          |     170
          2 | A          |     171
          2 | A          |    1071
(9 rows)
```

Routes have different route_i for each ramification. For example, there might be 8 different route_i for the RER A representing each branch in its plan. We had to merge them for the TP5 approach to work (otherwise there would have been more hops to account) and it stayed that way in the other programs. The result is in the table routexsuper.

CONCLUSION

DRAWBACKS

No primarykey/foreignkey defined in the schema because some nodes used in combined aren’t defined in nodes.

We wonder how to improve the app efficiency, as it already lags doing small computations.

The app doesn’t work very well compared to Google Maps but the path eventually brings you where you want to go relatively quickly. It ain’t absurd, just non optimal.

Our brain algorithm for clustering also uses name similarities between nodes in a cluster. We might have used string distances (Levenshtein distance) to build the clusters.

We should have done a distance based shortest path.

Printing the plan only is useful if you can do it within the City Mapper, but we had issue scaling the widget.

The docker image is super big and we are far from a Google Maps / City Mapper lightweight mobile app

We assume transport hours are the same everyday while they change in the weekend : we should have used temporal_week.

Conclusion

It was a very enjoyable project.

SQL revealed to be a useful tool for data inspection. A lot of queries can be emulated using pandas DataFrame but it is quicker and cleaner in SQL, even if it takes some line of code to query and fetch data with psycopg2.

Databases are also super useful to cache data.

PyQt5 GUI app development is relatively easy.

Docker makes it super simple to containerize a relatively complex app and distribute it.