The input data is given as a file with following structure:

- 1st line:
  → **number of junctions**
  → **number of streets**
  → **virtual time in seconds for the car itineraries**
  → **number of cars**
  → **start junction**

- Junctions
  → **longitude latitude** in decimal degrees (DD) ~ 41.40338, 2.17403

- Streets
  → **'from' junction index 'to' junction index directions time cost**

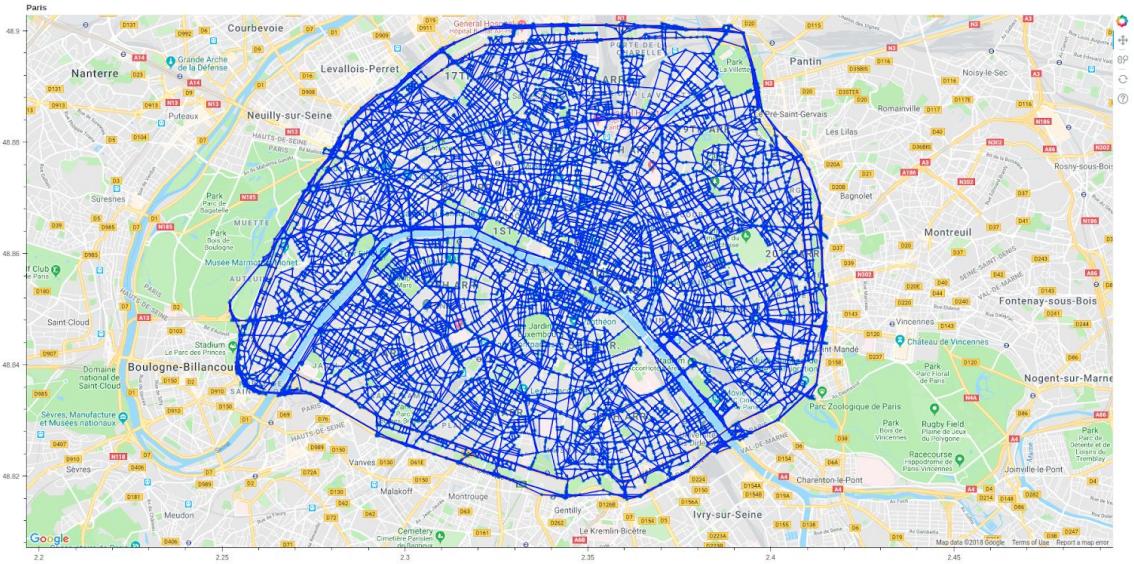
We used two ways of storing recieved data:

➔ working with csv file and using pandas dataframes for faster processing
➔ creating databases for quicker access to street info by junction's index + more convenient interface

As we need to store a lot of information about every street for further use in algorithm and it will be reasonable to be able to access it all by the enty-junction's index. For this purpose we decided to set up databases.

For different purposes/requests we'll have two tables:

1. **Visualization**
   ➔ pandas DataFrames to get coordinates by junction's index
   `df.iloc[j]["Longitude"]`
   ➔ bokah library for visualization
   ➔ GMap feature to add Paris background



2. Estimate Solution

➔ return data

- plain-text output file

number of cars
car 1… (iterary of visited streets)
car 2 …
…
car n ...

➔ requirements

- number of cars used for our algorithms must be ≤ initial number of cars stated in the input_data file
- all cars start from the “starting” junction stated in the input_data file
- all directions and streets must be valid and exist in given input
- cars can’t be beyond some junction when the time runs out

➔ scoring

- total length of visited streets (independently from amount of visits)

3. Storing data in Databases

➔ *Streets -- Table for all info about streets*

id	begin	end	Time	Len
…	...	...	...	...
int	int	int	int	int

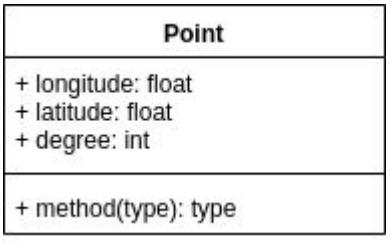
|auto-generated | Entry Point | End Point |

* if a street is two-directional there will be two rows for it in this table

➔ *Junctions -- Table for details about Junctions (Points)*

id	Latitude	Longitude	Degree
...	...	...	...
int	floats in decimal degrees		int

**



4. Algorithms

➔ **Greedy Algorithm**

Complete moving for each car, saving it's path into the 'path' attribute of Car class example. At the end write each car's movements is read from car's path and written into the result_file.

Choosing function is assigned to every car by [it's number % 2]
greedy functions:

- ➔ choose by max length + check whether street.time ≤ time_left
- ➔ choose by min time + check whether street.time ≤ time_left.

➔ **Euler Path Algorithm**