Report: Public Transport Project

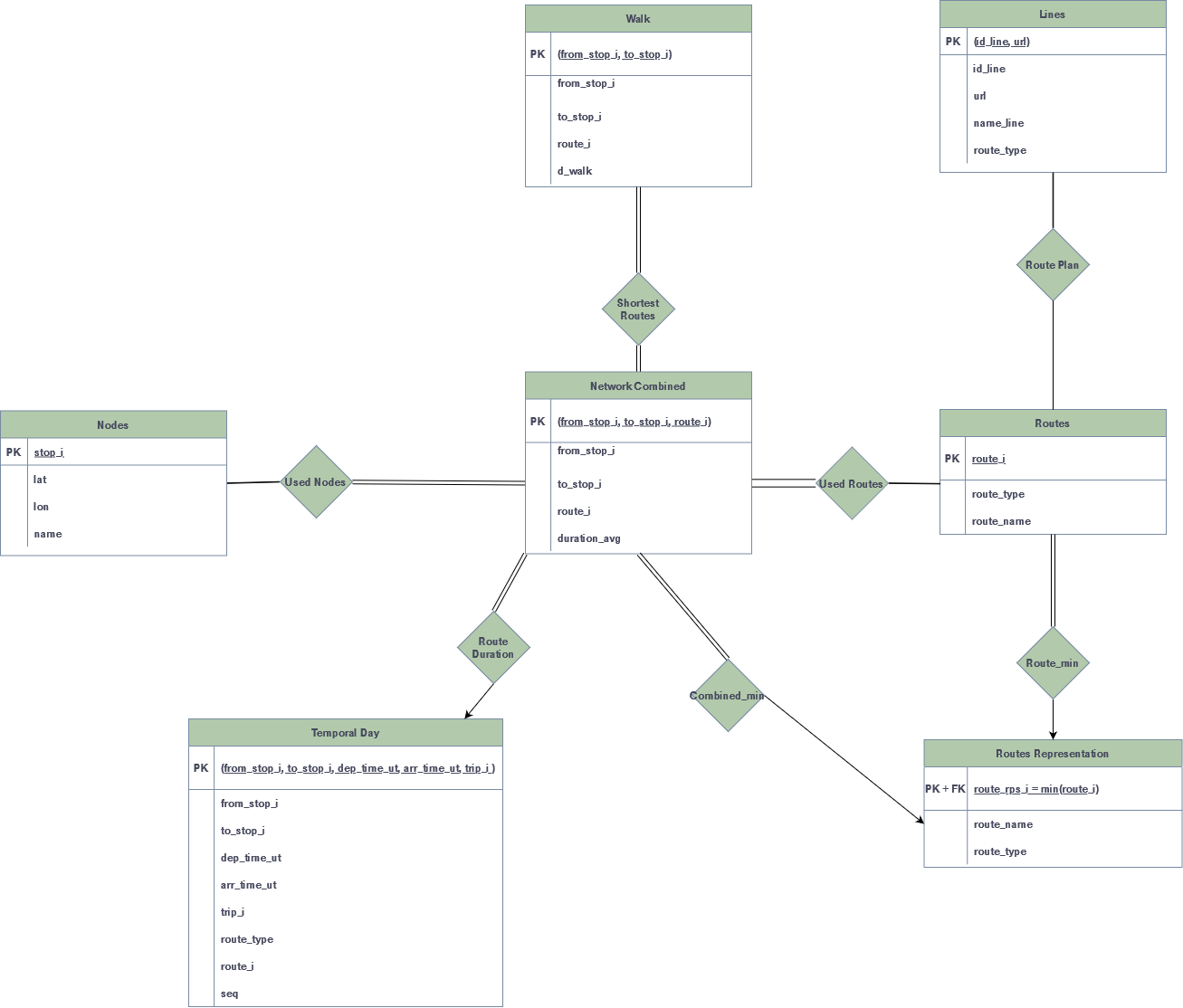
I – Functionalities list

A **requirements.txt** file has been generated to help the end-user install the modules necessary for this program. A **README.MD** file was written to clarify the installation and usage.

Given a dataset of public transport network data (.**csv** files containing network nodes and temporal data, .**geojson** files containing routes and stops names, all taken from <https://www.nature.com/articles/sdata201889>, as well as additional **.json** files taken from ) that is kept within a folder on the user’s computer, as well as a *PostgreSQL* server, our *PyQT5* application can:

* Prompt the user the path to the **data** folder (with a *FileDialog* that will be used to select the folder), as well as the login credentials necessary to connect to the postgres database (with a custom *QDialog* that has four entries: **user**, **password** (this one does not show plain text), **database** and **host** (the IP address which the postgres server runs on).
* Save the variables from previous step inside a “**params.json** file so they can be reused next time the program is launched. The json file, if already existing, is converted back into a set of variables by using the *json* module.
* Fill the postgres database with data obtained from the files within the **data** folder. The module *pandas* is used to convert the *.csv* files into a **DataFrame** (pandas’ way of storing tabular data). Said DataFrame is then used to create an equivalent *PostgreSQL* table. When necessary, we use the *psycopg2* module to run additional *SQL* requests that set the uniqueness of certain attributes in the table.
* Display a *QWindow* where the user can select a departure and a destination from two different Combo Boxes. Once both are selected, the user can press the “Go” button to launch the query that will give a path from the departure point to the arrival point. The user can also click the “Clear” button to clear the text within the Combo Boxes.
* Said window also features a *QTableWidget* that shows the detailed route the user must take to go from the departure point to the destination point on its first row, and the estimated time it would take on the second row. Clicking on a transport name (*RER E, BUS 165*…) within that table will make appear a message box, asking the user if they want to see said transport’s plan/map. If the user accepts, we will open the transport’s plan page within their default web browser. If there are multiple plans related to a transport name (often happens given how generic bus names are for example).
* Below that table, is a map (displayed with the *folium* module from a web page), which, when a route has been decided, shows all the stops of said route, linking each stop by a line, otherwise it shows all the stops that we know of on the map. The user can click a location on the map to select their departure and destination stops as well. All these actions are done by running *JavaScript* code with the *folium* module to interact with the map websites.
* Once the user has decided on a path to take, we use the *NetworkX* module to define the shortest path between the departure node and the arrival node. Given a set of *nodes* linked by *edges* (here routes), the module runs the *Dijkstra algorithm*, a graph known for its efficiency on finding the shortest path for road networks.

II – Entity-Relationship Diagram



The entities have been found with the following logic:

* The tables **temporal\_day, nodes, combined** and **walk** are based on the .*csv* files in the data folder.
* The table**route** is based on the .*geojson* file in the data folder.
* The table **line** is based on the *.json* files *fichers-horaires-et-plans* & *referentiel-des-lignes.*
* All the tables above have been imported by keeping the structure of their file:
* the keys for the tables based on the *.csv* files take the values of the first row within said files for each column that we need.
* The keys for the **route** table uses the *geojson* module to read the file as a dictionary and base the final table on the keys of said dictionary.
* The keys for the **line** table were generated in the same way, except that we merged both *json* dictionaries to only keep the data of transport lines that have a url attached to them.
* The table **route\_rps** has been created to keep in memory the “minimal” value a route can have (some routes have multiple *route\_i* values associated to them).

The **temporal\_day** file contains the following columns:

* *from\_stop\_i*: the departure stop of a route
* *to\_stop\_i*: the arrival stop of a route
* *dep\_time\_ut*: the Unix time at which we take the route from the departure stop
* *arr\_time\_ut*: the Unix at which we are supposed to arrive to the arrival stop
* *route\_type*: the transport ID that we use to go between two stops (0 is for trams, 1 is for metro, 2 is for trains, 3 is for buses… some other IDs exist but are not used in the Paris data, such as a Funicular ID or even a Boat ID)
* *trip\_I*: gives the ID given to a certain sequence of routes taken, **we won’t use it**
* *seq*: gives the sequence number within a trip, **we won’t use it**
* *route\_i: the ID given to a route (*i.e a possible path between two stops by using a certain transport)

This table is used to get the estimated duration of a trip between two stops. We can subtract the departure time from the arrival time to get a trip’s duration (in seconds).

The primary key of this table is the composite key **(from\_stop\_i, to\_stop\_i, dep\_time\_ut, arr\_time\_ut, route\_i)**:

* we can go from a stop to multiple other stops, which means having **from\_stop\_i** or **to\_stop\_i** alone isn’t enough to know any other key in the table.
* the same route can be used to have a trip between multiple stops, so having **route\_i** does not allow us to know the departure or the destination stops, but we can at least know the **route\_type**.
* We can take multiple routes to go from a stop to another, so the composite key (**from\_stop\_i, to\_stop\_i)** isn’t sufficient to know other keys in the table.
* knowing the departure time alone does not allow us to know which route was taken, nor where it was from or to, neither when the arrival time would be. Thus, knowing **dep\_time\_ut** or **arr\_time\_ut** alone wouldn’t be of much use.
* The composite key (**dep\_time\_ut, arr\_time\_ut)** doesn’t tell much either
* we can take a route from one stop to another at different times so having **(from\_stop\_i, to\_stop\_i, route\_i)** alone wouldn’t be enough.
* We can also have multiple routes assigned to a same time so using **(from\_stop\_i, to\_stop\_i, dep\_time\_u, arr\_time\_ut)** wouldn’t suffice.

And so, it seems that the only possible optimization in this table, is to guess the route\_type by knowing the other values already.

The functional dependencies are the following:

route\_i -> route\_type

from\_stop\_i, to\_stop\_i, route\_i, dep\_time\_ut, arr\_time\_ut -> route\_type

III – Tables & Dependencies

IV – Encountered Difficulties & Contributions per member