Université Paris Nanterre

Licence MIAGE

Rapport de projet
Graphes et Open data

Étude du réseau de transport RATP

Projet réalisé du 11 mars 2023 au 27 mai 2023

Membres du groupe

Badr AGRAD

Table des matières

1 Introduction

Du vendredi 26 juillet au dimanche 11 août, la France accueillera les Jeux Olympiques pour la XXXIIIe Olympiade. Plus de 15 millions de visiteurs sont attendus dans la capitale et sa métropole afin d'assister au plus grand évènement sportif au monde. Par conséquent, la gestion de l'affluence devient une priorité. En effet, l'accueil d'un nombre si élevé de touristes implique de répondre efficacement aux problématique de gestion des flux, de sécurité ou encore de déplacements.
La ville de Paris est organisée en 20 arrondissements. Pour faciliter les déplacements dans la capitale, elle dispose d'un réseau de transports en commun développé, comprenant notamment différentes lignes de métro, de RER, ainsi que de nombreux bus et tramways. Le métro parisien compte 16 lignes qui traversent la ville. Ces lignes sont numérotées de 1 à 14, avec deux lignes supplémentaires, 3bis et 7bis. Chaque ligne relie plusieurs stations stratégiques, permettant aux voyageurs de se rendre facilement d'un point à un autre de la ville. Le RER, est un réseau de train qui relie la ville de Paris à sa banlieue ainsi qu'à des destinations plus lointaines en Ile-de-France. La région dispose de quatre lignes de RER : la A, la B, la C ainsi que la D. En outre, Paris est entourée par le périphérique, une autoroute qui fait le tour de la ville et facilite les déplacements en voiture. Les principales gares ferroviaires de Paris, comme la Gare du Nord, la Gare de Lyon ou encore la Gare Montparnasse, sont également des hubs de transport importants, permettant des connexions avec le reste de l'hexagone ainsi que l'Europe.
Le réseau de transport RATP, englobant les lignes de metro et RER, est fréquenté quotidiennement par plus 6 millions de personnes, ce qui impose une fluidification du trafic ainsi qu'un agencement des lignes le plus optimal. Ces problématiques sont d'autant plus d'actualité dans le cadre des Jeux Olympique qui causera une forte augmentation de la fréquentation des transports en communs. Il est donc impératif de gérer les flux de touristes, d'avoir le maximum de stations opérationnelles ou encore de s'assurer des différentes correspondances.
J'ai donc entrepris de modéliser le trafic du réseau de transport RATP, incluant les lignes RER A et B, ainsi que les lignes de métro 1 à 14, et les lignes 3bis et 7bis. L'objectif de cette modélisation est d'analyser et de résoudre diverses problématiques liées à l'optimisation du réseau, à l'identification des points critiques, et à l'amélioration de la connectivité globale du réseau.

2 Environnement de travail

Ce projet a été réalisé en langage *Python* qui permet l'implémentation de différentes bibliothèques de graphe et de recherche opérationnelle.
Voici la liste des bibliothèques utilisées pour ce projet :

- Pandas : bibliothèque permettant la manipulation des fichiers csv
- NetworkX : bibliothèque permettant la création, la manipulation de graphes ainsi que certaines fonctions de recherche opérationnelle
- Matplotlib : bibliothèque permettant la visualisation des graphes et des résultats des analyses.
- osmnx : bibliothèque permettant de récupérer différents types de données géographiques. Elle est basée sur l'application OpenStreetMap, qui est une base de données cartographique open source
- Argparse : bibliothèque permettant la création d'interfaces en ligne de commande personnalisables pour les scripts Python.
- Random : bibliothèque permettant la génération de nombres aléatoire
- itertools : bibliothèque fournissant des outils pour créer et manipuler des itérables de manière efficace

Voici donc l'implémentation de toutes ces bibliothèque dans le code.

```
import pandas as pd
import networkx as nx
import matplotlib.pyplot as plt
import osmnx as ox
import argparse
```

L'ensemble de ces bibliothèques sont enregistrée dans un fichier `requirements.txt`.
Le projet a été développé sous environnement Unix.

3 Description du projet et problématiques

3.1 Les données utilisées

Pour modéliser le réseau, j'ai utilisé le fichier `trafic2021.csv` récupéré sur le site data.ratp.fr qui présente les données de trafic de l'année 2021 pour les lignes de transport RER A, RER B, les métros 1 à 14 ainsi que les lignes 3bis et 7 bis. Ce fichier contient des informations essentielles sur chaque station, y compris le trafic annuel et les correspondances disponibles. Voici un aperçu des premières lignes de ce fichier : Toutefois, on observe que le fichier `trafic2021` ne prend pas en compte l'ordre des

trafic2021

Rang	Réseau	Station	Trafic	Correspondance_1	Correspondance_2	Correspondance_3	Correspondance_4	Correspondance_5	Ville	Arrondissement pour Paris
6	Métro	BIBLIOTHEQUE	11104474	14					Paris	13.0
10	Métro	CHATELET	8350794	1	4	7	11.0	14.0	Paris	1.0
15	Métro	BOBIGNY-PABLO PICASSO	6561327	5					Bobigny	
17	Métro	GARE D'AUSTERLITZ	6318543	5	10				Paris	13.0

FIGURE 1 – un échantillon du fichier trafic2021.csv

stations dans une ligne.
Par exemple, supposons les stations ordonnées `A - B - C`. Cela implique qu'un lien doit exister entre les stations A et B ainsi que A et C mais pas entre A et C. En effet, une ligne doit forcément être représenté par un chemin ordonné.
Pour pallier à ce problème, j'ai créé un fichier externe `ordre.csv` qui contient, pour chaque ligne présente dans le fichier `trafic2021.csv`, les stations ordonnées. Certaines lignes, comme le RER A, ont des branches distinctes (par exemple, un itinéraire vers allant de Saint-Germain-en-Laye à Marne-la-Vallée Chessy et un autre se terminant à Boissy-Saint-Léger). Cette structuration m'a permis de modéliser un graphe représentant les voies ferrées comme des chemins. Voici un aperçu de ce fichier : Pour chaque

3	3BIS	4
PONT DE LEVALLOIS-BECON	PORTE DES LILAS	PORTE DE CLIGNANCOURT
ANATOLE FRANCE	SAINT-FARGEAU	SIMPLON
LOUISE MICHEL	PELLEPORT	MARCADET-POISSONNIERS
PORTE DE CHAMPERRET	GAMBETTA	CHATEAU ROUGE
PEREIRE		BARBES-ROCHECHOUART
WAGRAM		GARE DU NORD
MALESHERBES		GARE DE L'EST
VILLIERS		CHATEAU D'EAU

FIGURE 2 – un échantillon du fichier ordre.csv

station, j'ai décidé de récupérer la position géographique (latitude et longitude). En effet, cela pouvait être intéressant de procéder à un algorithme de placement. Cela rend permet au graphe de représenter au mieux la réalité. A l'aide de la bibliothèque osmnx, j'ai codé une fonction qui réalise une requête OpenStreetMap http avec le nom de la Station ainsi que la ville.

```
import requests

def get_coordinates(station_name, city, district=None):
    query = f'{station_name} {city}'
```

```
    if district:
        query += f' {district}'
    url = f'https://nominatim.openstreetmap.org/search?q={query}
    &format=json'
    response = requests.get(url)
    data = response.json()
    if data:
        return (data[0]['lat'], data[0]['lon'])
    else:
        return None
```

Cependant, cette méthode ne fonctionnait que pour les stations situées à Paris. Pour les stations situées dans d'autres villes (par exemple, Cergy-le-Haut), j'ai dû récupérer manuellement les coordonnées sur le site de la RATP. Voici un aperçu des premières lignes du fichier `trafic2021.csv` avec les colonnes de coordonnées ajoutées (certaines colonnes vu précédemment on été masquée par soucis de visibilité) :

Réseau	Station	Trafic	Coordonnées	Latitude	Longitude
Métro	BIBLIOTHEQUE	11104474	48.829990199899186, 2.375747983215603	48.829990199899186	2.375747983215603
Métro	CHATELET	8350794	48.8601084, 2.3464367	48.8601084	2.3464367
Métro	BOBIGNY-PABLO PICASSO	6561327	48.9063747711693, 2.4491899044793364	48.9063747711693	2.4491899044793364
Métro	GARE D'AUSTERLITZ	6318543	48.8412453, 2.3663585	48.8412453	2.3663585
Métro	HAVRE-CAUMARTIN	5894982	48.8736792, 2.3273189	48.8736792	2.3273189
Métro	AUBERVILLIERS-PANTIN-QUATRE CHEMINS	5616435	48.90378388105197, 2.392221222535117	48.90378388105197	2.392221222535117

FIGURE 3 – un échantillon du fichier position.csv

3.2 Les problématiques de recherche opérationnelles

Une fois le dataSet prêt à l'emploi, j'ai décidé de répondre à trois problématiques de recherche opérationnelles.

— Bloquer le trafic en bloquant un nombre minimum de stations (deux approches ont été testées)

— Identifier et analyser l'impact des stations "congestionnées" sur le réseau de transport

— Trouver le plus court chemin pour visiter une liste de monuments dans Paris

Différents algorithmes de RO seront utilisés afin de répondre à ces problèmes.

4 Travail réalisé

4.1 Les graphes

Avec les noms des stations les positions géographiques exactes des stations ainsi que l'ordre, nous obtenons un graphe qui propose un chemin ordonné entre les stations. De plus, ce graphe prend en compte les différentes correspondances, connexions. Considérons donc que si une station `A` est présente sur la ligne 1, 5 et 14, un lien sera crée entre les trois lignes avec pour sommet commun la station `A`. Ce graphe représente les stations de transport avec leurs connexions, incluant les correspondances multiples. Cette modélisation précise permet de visualiser et d'analyser le réseau.

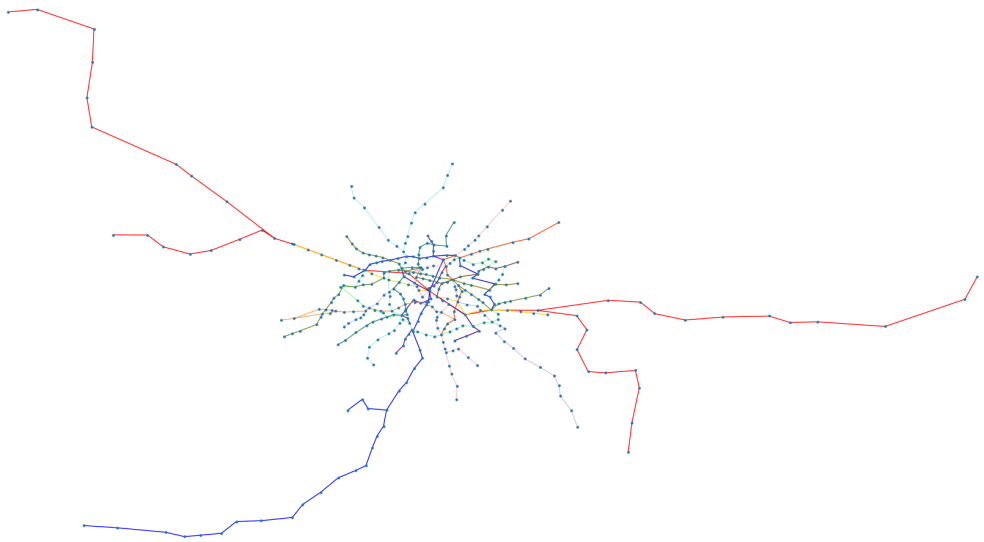

FIGURE 4 – un échantillon du fichier ordre.csv

4.2 La recherche opérationnelle

4.2.1 Première problématique

L'objectif est de déterminer quelles stations doivent être bloquées pour interrompre complètement le trafic sur le réseau de transport en utilisant le nombre minimum de stations. Cette problématique est cruciale pour comprendre les vulnérabilités du réseau et identifier les points critiques qui, s'ils sont perturbés, peuvent causer une interruption majeure du service. Pour résoudre ce problème, nous Allons étudier deux approche différente.
La première consiste à utiliser le concept de "coupe minimale de sommets" dans un graphe. Une "coupe de sommets" est un ensemble de sommets (stations) dont la suppression déconnecte le graphe, c'est-à-dire empêche toute communication entre certains nœuds. La "coupe minimale de sommets" est le plus petit ensemble de sommets qui, une fois supprimé, déconnecte le graphe.

Algorithm 1 Trouver Coupe Minimale de Sommets

```
function TROUVER_COUPE_MINIMALE_DE_SOMMETS(G)
    min_cut ← {}
    min_cut_size ← infini
    for chaque paire de sommets (u, v) dans G do
        // Calculer le flot maximum de u à v
        flot_max, coupe ← CALCULER_FLOT_MAX_ET_COUPE(G, u, v)
        if taille de coupe < min_cut_size then
            min_cut ← coupe
            min_cut_size ← taille de coupe
        end if
    end for
    return min_cut
end function
```

Algorithm 2 Calculer Flot Max et Coupe

```
function CALCULER_FLOT_MAX_ET_COUPE(G, u, v)
    // Initialiser le flot à 0 pour toutes les arêtes
    initialiser flot à 0 pour chaque arête dans G
    // Algorithme de flot maximum (Edmonds-Karp, par exemple)
    flot_max ← 0
    repeat
        trouver chemin augmentant p de u à v
        augmenter flot le long de p
        flot_max ← flot_max + flot de p
    until il n'y a plus de chemin augmentant de u à v
    // Trouver la coupe minimale associée au flot maximum
    coupe ← ensemble de sommets atteignables depuis u dans le graphe résiduel
    return flot_max, coupe
end function
```

— **Initialisation** : Nous commençons par définir `min_cut` comme un ensemble vide et `min_cut_size` comme infini.
— **Boucle sur toutes les paires de sommets** : Pour chaque paire de sommets (u, v) dans le graphe, nous calculons le flot maximum et la coupe associée en utilisant l'algorithme de flot maximum (comme Edmonds-Karp).
— **Calcul du flot maximum et de la coupe** : Nous initialisons le flot à 0 pour toutes les arêtes. Nous utilisons l'algorithme de flot maximum pour trouver le flot maximum de u à v. Cet algorithme repose sur la recherche de chemins augmentant et l'augmentation du flot le long de ces chemins. Une fois le flot maximum trouvé, nous identifions la coupe minimale associée, qui est l'ensemble des sommets atteignables depuis u dans le graphe résiduel.
— **Mise à jour de la coupe minimale** : Si la taille de la coupe trouvée est inférieure à `min_cut_size`, nous mettons à jour `min_cut` et `min_cut_size`.
— **Retour de la coupe minimale** : Après avoir itéré sur toutes les paires de sommets, nous retournons l'ensemble `min_cut` qui représente la coupe minimale de sommets.

L'approche basée sur l'algorithme de Flot Maximum - Coupe Minimum est théoriquement solide pour identifier les points critiques dans un réseau de transport. Cependant, sa complexité peut augmenter de manière drastique pour des graphes très grands, car elle nécessite de calculer le flot maximum pour chaque paire de sommets. La complexité de l'algorithme de Ford-Fulkerson avec l'implémentation d'Edmonds-Karp est de $O(E \cdot V^2)$, où E est le nombre d'arêtes dans le graphe et V est le nombre de sommets. Cependant, il est important de noter que cette complexité peut être trompeuse, car dans le pire des cas, l'algorithme peut nécessiter un grand nombre d'itérations, ce qui peut entraîner une complexité en temps exponentielle. C'est pourquoi dans certaines situations, l'algorithme de Ford-Fulkerson peut être inefficace. Dans certains cas, des approximations ou des heuristiques peuvent être nécessaires pour rendre l'approche plus efficace.
Voici le résultat obtenu à l'aide de cette première approche :

```
python3.11 RO.py -bloquertrafic
    Stations à bloquer: ['GARE DE LYON']
    Nombre de stations à bloquer: 1
```

Nous constatons que `GARE DE LYON` est une station qui comprend trois correspondances : Le RER A, metro 1, le metro 14. Toutefois, il n'est pas très réaliste d'envisager que le blocage de Gare de Lyon bloquerait l'ensemble du réseau de transport RATP. Ce résultat pourrait être du au fait qu'il manque certaines informations qui aurait pu améliorer l'algorithme de flot. En effet, les arrêtes ne sont pas pondérées (ou toute pondérée a 1) ce qui a pu perturber l'algorithme.

La seconde approche se base sur le concept de degré d'un sommet. Le degré d'un sommet correspond au nombre d'arêtes connectées à ce sommet. L'objectif de cette approche sera de déterminer l'ensemble des degrés des sommets du graphe et de sélectionner le minimum de sommets avec le degrés le plus élevé.

L'approche basée sur les degrés des stations dans le réseau est une méthode couramment utilisée pour identifier les points critiques. En effet, les stations avec le plus grand nombre de connexions sont souvent considérées comme les plus critiques, car leur congestion peut avoir un impact significatif sur l'ensemble du réseau. De plus cette approche propose une complexité moins élevé que l'approche basée sur le flot maximum. Plutôt que de calculer des flux à travers le réseau, elle se concentre uniquement sur

Algorithm 3 bloquer_trafic2(G)

1: *station_degrees* ← dictionnaire des degrés des stations dans G
2: *sorted_stations* ← trier les stations par degré décroissant
3: *top_critical_stations* ← les 10 premières stations de *sorted_stations*
4: **print** "Stations critiques :", *top_critical_stations*
5: **print** "Nombre de stations critiques :", **longueur**(*top_critical_stations*)

les connexions directes entre les stations. Néanmoins, l'approche basée sur les degrés peut également présenter des limites. Par exemple, elle peut ne pas prendre en compte certains facteurs importants comme la capacité des lignes ou les différences de trafic entre les différentes parties du réseau. De plus, elle peut surestimer l'importance de certaines stations simplement en fonction de leur connectivité, sans tenir compte d'autres aspects tels que la localisation géographique.
Voici le résultat obtenu à l'aide de cette première approche :

```
Top critical stations: ['NATION', 'CHATELET', 'REPUBLIQUE',
'MONTPARNASSE-BIENVENUE', 'GARE DE LYON', 'BASTILLE', 'SAINT-
LAZARE', 'OPERA', 'LA MOTTE-PICQUET-GRENELLE', 'CHARLES DE GAULLE-
ETOILE']
Number of top critical stations: 10
```

Nous constatons cette fois ci que nous obtenons un résultat davantage réaliste. En effet, les stations présente dans cette liste comportent toutes un grand nombre correspondances. Voici le graphe avec les stations critiques en surbrillance (rouge).

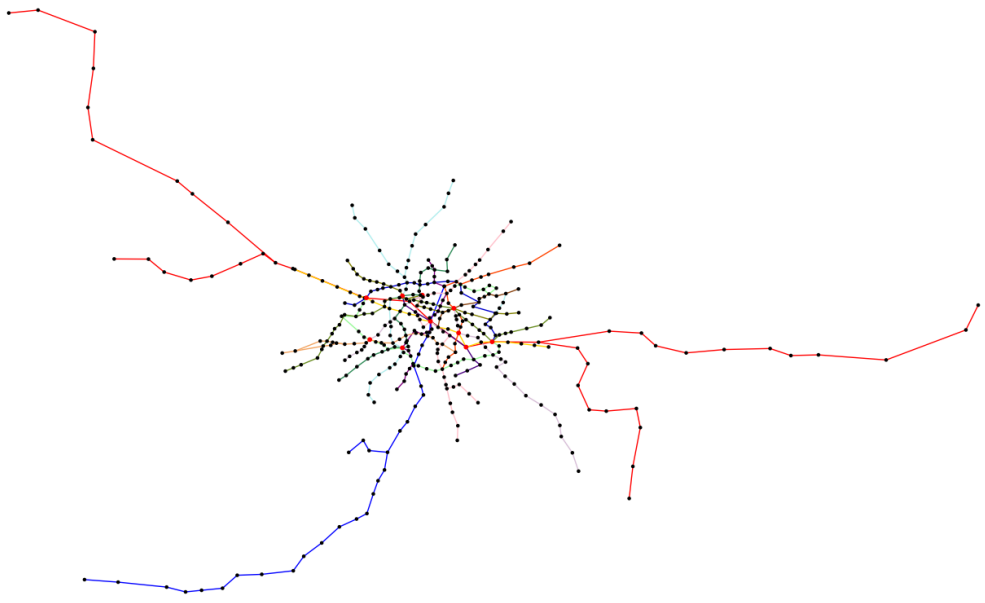

FIGURE 5 – Le graphe avec les stations critiques en Rouge

4.2.2 Deuxième problématique

L'objectif est d'identifier les stations les plus congestionnées dans un réseau de transport (celles avec le plus grand nombre de correspondances) et d'analyser l'impact de la suppression de ces stations sur la connectivité globale du réseau. Cette analyse permet de comprendre comment certaines stations peuvent devenir des points de congestion et comment leur perturbation affecte la performance du réseau. Nous allons donc observer la différence de longueur de la moyenne des chemins les plus courts (en termes de nombre de stations) avant et après la suppression des stations congestionnées.
Pour résoudre ce problème, nous utilisons une approche basée sur le degré des sommets dans le graphe pour identifier les stations congestionnées. Ensuite, nous calculons la longueur moyenne des plus courts chemins avant et après la suppression de ces stations pour mesurer leur impact sur la connectivité du réseau.

Algorithm 4 Path de Congestion

```
function CONGESTION_PATH(G)
    degree_dict ← dictionnaire vide
    for chaque station dans G do
        degree_dict[station] ← degré de la station
    end for
    max_degree ← valeur maximale de degree_dict
    congested_stations ← liste des stations avec degree_dict[station] == max_degree
    original_avg_path_length ← calculer_longueur_moyenne_plus_courts_chemins(G)
    G_removed ← copier G
    supprimer congested_stations de G_removed
    avg_path_lengths ← liste vide
    connected_components ← trouver_composants_connectés(G_removed)
    for chaque composant dans connected_components do
        subgraph ← sous-graphe de G_removed pour le composant
        if taille de subgraph > 1 then
            avg_path_lengths.append(calculer_longueur_moyenne_p
lus_courts_chemins(subgraph))
        end if
    end for
    if avg_path_lengths n'est pas vide then
        new_avg_path_length ← moyenne de avg_path_lengths
    else
        new_avg_path_length ← infini
    end if
end function
```

- **Identifier les stations congestionnées** : Nous calculons le degré de chaque station, c'est-à-dire le nombre de connexions qu'elle a avec d'autres stations. Nous identifions les stations ayant le degré maximal, car elles sont considérées comme les plus congestionnées.
- **Calculer la longueur moyenne des plus courts chemins avant suppression** : Nous calculons la longueur moyenne des plus courts chemins dans le graphe initial pour avoir une base de comparaison.
- **Supprimer les stations congestionnées** : Nous créons une copie du graphe et supprimons les stations congestionnées.
- **Calculer la longueur moyenne des plus courts chemins après suppression** : Nous trouvons les composants connectés restants dans le graphe sans les stations congestionnées. Nous calculons la longueur moyenne des plus courts chemins pour chaque composant connecté. Nous prenons la moyenne de ces longueurs pour obtenir la nouvelle longueur moyenne après suppression.
- **Visualisation** : Nous affichons le graphe complet avec les stations congestionnées en rouge pour visualiser les points de congestion. Nous affichons les sous-graphes résultants sans les stations congestionnées, colorés différemment pour chaque composant, afin de visualiser l'impact de leur suppression.

Cette approche est utile pour identifier les points de congestion dans un réseau de transport et pour comprendre l'impact de la suppression de ces points sur la connectivité du réseau. Cependant, elle se base uniquement sur le degré des stations, ce qui peut ne pas capturer toutes les dynamiques de congestion, telles que les flux de passagers ou

les horaires de pointe.
Voici le résultat obtenu :

```
Stations les plus congestionnées (degré = 8): ['NATION', 'CHATELET',
'REPUBLIQUE', 'MONTPARNASSE-BIENVENUE']
Longueur moyenne des plus courts chemins avant suppression:
13.95116607570123
Longueur moyenne des plus courts chemins après suppression:
6.282647624279044
```

Nous constatons que la longueur moyenne des plus courts chemins après la suppression des stations congestionnés est inférieur à la moyenne avant la suppression de celle ci. En effet, cela est dû au fait que la suppression de ces stations déconnecte le graphe est génère des sous-graphe. Par conséquent, cette moyenne correspond aux sous graphes puisqu'il est impossible de calculer pour les graphes (la moyenne serait `inf`).
Nous pouvons constater sur cette image la déconnexion du graphe a différents emplacements.

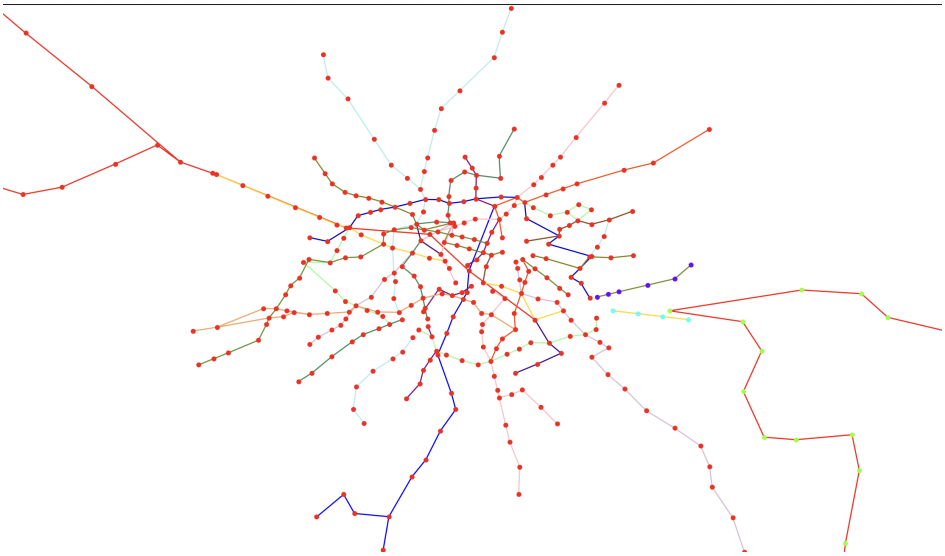

FIGURE 6 – Le graphe avec les sous graphes

4.2.3 Troisième problématique

L'objectif est de déterminer le chemin le plus court (en termes de nombre de stations) pour visiter une liste de monuments touristiques dans un réseau de transport. Cette problématique est essentielle pour optimiser les trajets touristiques, permettant de minimiser le temps et les efforts nécessaires pour visiter plusieurs sites d'intérêt.

Pour résoudre ce problème, nous utilisons une approche basée sur l'algorithme de permutation des monuments pour évaluer tous les chemins possibles, puis nous sélectionnons le chemin avec la longueur totale minimale. Cette méthode est basée sur le problème du voyageur de commerce (TSP - Travelling Salesman Problem), une problématique classique d'optimisation combinatoire.

Algorithm 5 Trouver Plus Court Chemin

```
function TROUVER_PLUS_COURT_CHEMIN(G, liste_monuments)
    meilleur_chemin ← None
    longueur_minimale ← infini
    for chaque permutation des monuments do
        longueur_actuelle ← 0
        chemin_actuel ← liste vide
        for chaque paire consécutive de monuments dans la permutation do
            chemin_segment ← TROUVER_CHEMIN_COURT(G, paire[0], paire[1])
            longueur_actuelle ← longueur_actuelle + longueur(chemin_segment) - 1
  ▷ Soustraire 1 pour éviter de compter deux fois les intersections
            ajouter chemin_segment sans le dernier nœud à chemin_actuel
        end for
        ajouter le dernier monument de la permutation à chemin_actuel
        if longueur_actuelle < longueur_minimale then
            longueur_minimale ← longueur_actuelle
            meilleur_chemin ← chemin_actuel
        end if
    end for
    return meilleur_chemin, longueur_minimale
end function
```

Algorithm 6 Trouver Chemin Court

```
function TROUVER_CHEMIN_COURT(G, station_source, station_cible)
    initialiser une file de priorité avec la station_source
    initialiser un dictionnaire pour stocker les distances depuis la station_source
    initialiser un dictionnaire pour stocker les précédents nœuds dans le chemin
    while la file n'est pas vide do
        extraire le nœud avec la plus petite distance
        for chaque voisin du nœud actuel do
            calculer la nouvelle distance
            if la nouvelle distance est plus petite then
                mettre à jour la distance et le précédent nœud
                ajouter le voisin à la file de priorité
            end if
        end for
    end while
    reconstruire le chemin en utilisant les précédents nœuds
    return le chemin
end function
```

- **Initialisation** : Nous commençons par définir `meilleur_chemin` comme `None` et `longueur_minimale` comme infini.
- **Génération de permutations** : Pour chaque permutation des monuments à visiter, nous calculons la longueur totale du chemin en visitant chaque monument dans l'ordre de la permutation.
- **Calcul du chemin pour chaque permutation** : Pour chaque paire consécutive de monuments dans la permutation, nous utilisons un algorithme de plus court chemin pour trouver le chemin entre eux. Nous soustrayons 1 de la longueur du segment pour éviter de compter deux fois les intersections. Nous ajoutons chaque segment de chemin au `chemin_actuel`.
- **Mise à jour du meilleur chemin** : Si la longueur actuelle est inférieure à la longueur minimale trouvée jusqu'à présent, nous mettons à jour `meilleur_chemin` et `longueur_minimale`.
- **Retour du meilleur chemin** : Après avoir évalué toutes les permutations, nous retournons le chemin avec la longueur totale minimale.

Cette approche est exhaustive et garantit de trouver le chemin optimal en termes de longueur totale. Cependant, elle peut comporter une complexité assez élevé, surtout si il y a un grand nombre de monuments à visiter, car le nombre de permutations augmente de manière factorielle. Dans de tels cas, des heuristiques ou des algorithmes approximatifs peuvent être utilisés pour trouver des solutions assez proches de l'optimal en moins de temps.
Voici le résultat obtenu :

```
Starting station: INVALIDES
Shortest path with lines:
INVALIDES (Lignes Unknown)
CHAMPS-ELYSEES-CLEMENCEAU (Lignes Unknown)
FRANKLIN D. ROOSEVELT (Lignes Unknown)
ALMA-MARCEAU (Lignes Unknown)
```

```
IENA (Lignes Unknown)
TROCADERO (Lignes Unknown)
IENA (Lignes Unknown)
ALMA-MARCEAU (Lignes Unknown)
FRANKLIN D. ROOSEVELT (Lignes Unknown)
CHAMPS-ELYSEES-CLEMENCEAU (Lignes Unknown)
CONCORDE (Lignes Unknown)
MADELEINE (Lignes Unknown)
OPERA (Lignes Unknown)
PYRAMIDES (Lignes Unknown)
PALAIS-ROYAL (Lignes Unknown)
LOUVRE (Lignes Unknown)
CHATELET (Lignes Unknown)
GARE DE LYON (Lignes Unknown)
CHATELET-LES HALLES-RER (Lignes Unknown)
SAINT-MICHEL-NOTRE-DAME (Lignes Unknown)
CHATELET-LES HALLES-RER (Lignes Unknown)
AUBER (Lignes Unknown)
CHARLES DE GAULLE-ETOILE-RER (Lignes Unknown)
Number of stations in shortest path: 22
```

FIGURE 7 – Le chemin le plus court sous forme de graphe

5 Conclusion

In fine, les résultats de ce projet sont assez cohérents mais à mon sens peuvent manquer de précision et, dans certain cas, de réalisme. En effet, le dataset n'étant pas assez optimisé, j'ai du passer beaucoup de temps à rechercher des données ainsi qu'à les trier. L'absence d'une partie du trafic (RER D, RER C, certaines stations des RER A et B, bus, tramway...) a aussi joué dans cette étude. Je pense que la partie la mieux réussie est celle du chemin le plus court pour parcourir la liste des monuments incontournables de Paris. D'un autre côté, j'ai trouvé le résultat de la première problématique assez peu convaincant, notamment lors de sa résolution avec l'algorithme des Flots. J'ai apprécié travailler en Python car ce langage propose énormément de modules facilitant le travail. Ainsi, cela permet de se concentrer au maximum sur les problématiques.

6 Rétrospective de mon travail

A cours de ce projet, j'ai eu l'opportunité de comprendre que la Data est un élément plus que nécessaire. En effet, des data incomplètes ou défaillantes peut complètement fausser les résultats. Il est essentiel de réaliser un travail conséquent de récupération et de filtrage des données que l'on souhaite exploiter. De plus, il est important de ne pas négliger des informations qui, au premier abord, peuvent nous paraître impertinentes. Chaque détail compte et peut apporter une nuance à notre résultat.
Effectivement, j'ai passé beaucoup de temps à nettoyer les données et a améliorer la pertinence du dataset. Avec plus de temps ainsi qu'une meilleure organisation, je pense que j'aurais pu récupérer davantage de données pertinentes, apportant une exactitude et un réalisme à mes résultats.
J'ai trouvé que la phase de recherche de sujet et de problématiques était la plus fastidieuse. Il m'est arrivé à de nombreuses reprises de trouver des datasets qui paraissaient intéressants mais pour lequel dégager des problèmes de recherche opérationnelles était très compliqué.
Toutefois, l'usage des intelligences artificielle génératives comme Chat-GPT 4 et son module network intelligence, spécialisé dans les graphes, ou encore Mistral m'ont permis un gain de temps ainsi qu'une optimisation de mon code. A l'avenir, je pense qu'il est nécessaire d'automatiser au maximum toute la partie liée au nettoyage et à la récupération de fichiers pour me concentrer davantage sur la recherche opérationnelle.
En somme, j'ai trouvé ce premier projet de recherche opérationnelle très intéressant. J'ai pu témoigner de la puissance de certains algorithme et de l'utilité de la théorie des graphes. Je souhaite continuer à m'investir dans l'étude des données et l'optimisation et améliorer ma qualité de rendement sur ce type de projets.

7 Webographie

Références

[le site Open Data de la RATP] https://data.ratp.fr/explore/?sort=modified

[Documentation de Mistral] https://networkx.org/documentation

[Théorie des graphes] https://fr.wikipedia.org/wiki/Th%C3%A9orie_des_graphes

[La recherche opérationnelle] https://fr.wikipedia.org/wiki/Recherche_op%C3%A9rationnelle

8 Manuel utilisateur

D'une part, il est nécessaire de disposer de l'ensemble des bibliothèques et des dépendances.
Ainsi, entrez la commande : `pip install -requirements.txt` dans le dossier.
Le fichier a exécuter se nomme `RO.py` Chaque problématique du fichier est défini par un argument. L'exécution sans argument vous propose les différentes possibilités.

```
python3.11 RO.py
Choisir un argument pour l'exécution du code.
-graphe : affiche le graphe du trafic ratp
-bloquertrafic : Bloquer un nombre de stations minimum pour bloquer
l'ensemble du trafic, Flot Maximum - Coupe Minimale
- bloquertrafic2: Bloquer un nombre de stations minimum pour bloquer
l'ensemble du trafic, degrés des stations
-touriste : propose un plus court chemin afin de visiter les différents
grands monuments parisiens
-congestionpath : trouve les stations les plus congestionnées et calcule
la moyenne des plus courts chemins avant et après leur suppression.
```

Voici les cinq exécutions possibles :

```
python3.11 RO.py -graphe
pyhon3.11 RO.py -bloquertrafic
python3.11 RO.py -bloquertrafic2
python3.11 RO.py -touriste
python3.11 RO.py -congestionpath
```