ADIL CHETOUANI
MEHEDI TOURE
GODWIN KANLINSOU

Rapport de Projet
“Vas-y dans le métro”

2024

Introduction

Contexte du projet

Ce projet utilise la théorie des graphes pour modéliser le réseau de métro parisien. Le réseau est représenté comme un graphe non orienté, et plusieurs algorithmes sont implémentés pour résoudre des problèmes pratiques liés aux trajets et à la structure du réseau. On rappelle que le graphe est non orienté bien que certaines lignes possèdent des orientations, notamment la ligne 10 qui sera simplifiée.

Rappel des objectifs principaux

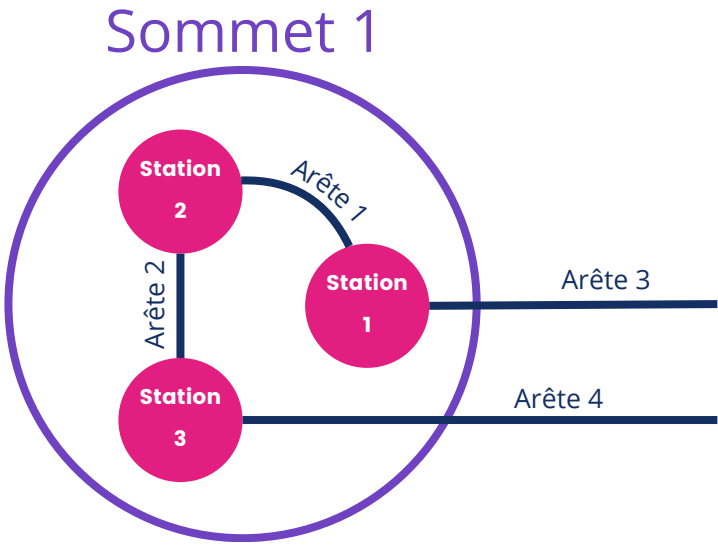
Les trois objectifs principaux sont :

- Vérification de la connexité : Assurer que toutes les stations sont accessibles.
- Calcul du plus court chemin : Trouver le chemin le plus court entre deux stations avec l'algorithme de Bellman-Ford et une estimation du temps.
- Extraction de l'arbre couvrant de poids minimum (ACPM) : Créer un sous-ensemble optimisé de liaisons entre stations avec l'algorithme de Prim.

Structure du project

Pour notre projet, nous modélisons le réseau de métro parisien à partir du fichier metro.txt, où les sommets représentent des groupes de stations et les arêtes correspondent aux voies de métro.

Un sommet est en outre est sous-graphe du graphe.



Méthodologie

Choix du langage de programmation

Les bibliothèques standard de Python sont utilisées pour la gestion des structures de données et des algorithmes pas de framework.

Pour la partie visualisation en a opté pour la libraire NetworkX.

Pour le bonus on a utilisé

- Python pour les algorithmes,
- Flask un framework pour faire les api, (Back)
- Html, Css et Js pour UI (Front)

Structure des Fichiers

Le dossier entity va avoir tous les objets relatifs à la théorie des graphes (arête, sommet, graphe...)

Arete.py
Sommet.py
Station.py
Graphe.py

entity

Le dossier res contiendra toutes les fichiers externes au projet comme les CSV de données.

graph.png
metro.txt
metrof_r.png
pospoints.txt

res

Le dossier de service contiendra toutes les fonctionnalités liées à l'algorithme et à l'affichage.

algorithme.py
draw_graph.py
read_file.py

service

Structure des données

Nous avons utilisé une approche orientée objet, qui nous a permis de capturer les caractéristiques et les relations des stations de métro, des lignes, et des connexions.

Nous avons créé plusieurs classes interconnectées : Graphe, Sommet, Station et Arete. Chaque classe joue un rôle spécifique pour organiser les informations et simplifier les opérations sur le graphe.

1. Classe Graphe

La classe Graphe est le cœur de notre projet. Elle contient les sommets (stations) et les arêtes (connexions entre les stations). Elle fournit des méthodes pour gérer et interroger le graphe.

```
class Graphe:
    def __init__(self):
        self.sommets = []  # Liste des sommets
        self.aretes = []   # Liste des arêtes
```

2. Classe Sommet

La classe Sommet représente un groupe de stations de métro associées grâce au nom et aux coordonnées géographiques.

```
class Sommet:
    def __init__(self, nom_sommet):
        self.nom_sommet = nom_sommet
        self.stations = []  # Liste des objets Station
        self.pos = []       # Coordonnées (x, y) des stations
```

Structure des données

3. Classe `Station`

La classe Station contient des détails comme le numéro de ligne, si la station est un terminus et sont numéro de branchements.

```python
class Station:
    def __init__(self, num_sommet, numero_ligne, si_terminus=False, branchement=0):
        self.num_sommet = num_sommet # Numéro du sommet
        self.numero_ligne = numero_ligne  # Numéro de la ligne
        self.si_terminus = si_terminus  # Si la station est un terminus (booléen)
        self.branchement = branchement  # Indicateur de branchement (0, 1, 2, etc.)
```

4. Classe `Arete`

La classe Arête représente une connexion entre deux stations avec le temps de parcours.

```python
class Arete:
    def __init__(self, num_station1, num_station2, time_sec, sommet1=None, sommet2=None):
        self.num_station1 = num_station1 # Numéro de la station 1
        self.num_station2 = num_station2 # Numéro de la station 2
        self.time_sec = time_sec # Temps en secondes pour parcourir l'arête

        self.sommet1 = sommet1 # Sommet 1
        self.sommet2 = sommet2  # Sommet 2
```

Algorithmes implémenté

1. Vérification de la Connexité avec BFS

L'algorithme de parcours en largeur (BFS) est utilisé pour vérifier que le graphe est connexe. Si un sommet ne peut pas être atteint, des arêtes supplémentaires sont nécessaires pour rendre le graphe connexe.

Entrée : Le graphe `graph` et un sommet de départ `start_sommet`.
Sortie : Un ensemble de sommets visités, permettant de vérifier si tous les sommets sont accessibles depuis le sommet de départ.

```python
def bfs(graph: Graphe, start_sommet: Sommet):
    visited = set()  # Ensemble des sommets visités
    queue = [start_sommet]  # File d'attente pour gérer les sommets à visiter

    while queue:
        sommet = queue.pop(0)  # Retirer le premier élément de la file
        if sommet not in visited:
            visited.add(sommet)  # Marquer le sommet comme visité
            adjacents = graph.get_sommet_adjacents(sommet)  # Obtenir les voisins

            for adjacent in adjacents:
                if adjacent not in visited:
                    queue.append(adjacent)  # Ajouter les voisins non visités à la file

    return visited
```

Le code réalise un parcours en largeur (BFS) dans notre graphe à partir d'un sommet de départ (start_sommet). Il utilise une file d'attente (queue) pour stocker les sommets à explorer et un ensemble (visited) pour suivre les sommets déjà visités, évitant ainsi les doublons. À chaque itération, le sommet en tête de file est traité : il est marqué comme visité, puis ses voisins (sommets adjacents) non visités sont ajoutés à la file d'attente. La fonction renvoie finalement tous les sommets visités, représentant les sommets atteignables depuis le point de départ.

Algorithmes implémenté

2. Calcul du Plus Court Chemin avec Bellman-Ford

L'algorithme de Bellman-Ford est utilisé pour calculer le plus court chemin, tenant compte des temps de parcours entre stations.

```python
def bellman_ford(graph, station):
    distances = {station.num_sommet: float('inf') for station in graph.get_stations()}
    distances[station.num_sommet] = 0
    predecesseur = {station.num_sommet: None for station in graph.get_stations()}

    for _ in range(len(graph.sommets) - 1):
        for arete in graph.aretes:
            u = arete.num_station1
            v = arete.num_station2
            poids = arete.time_sec

            if distances[u] != float('inf') and distances[u] + poids < distances[v]:
                distances[v] = distances[u] + poids
                predecesseur[v] = (u, arete.sommet1)

    return distances, predecesseur
```

Pour l'algorithme de Bellman-Ford qui sert à trouver les plus courtes distances depuis une station de départ dans un graphe pondéré. Il initialise les distances des stations à l'infini, sauf celle de départ (0), et associe un prédécesseur à chaque station.
Ensuite, pour chaque arête du graphe, il vérifie si passer par une station intermédiaire réduit la distance vers une autre, et met à jour les distances et prédécesseurs en conséquence. Après les relaxations, il renvoie les distances minimales et les prédécesseurs, permettant ainsi de déterminer les chemins les plus courts depuis la station initiale pour toutes les stations.

Algorithmes implémenté

3. Arbre Couvrant de Poids Minimum (ACPM) avec Prim

L'algorithme de Prim est implémenté pour trouver l'arbre couvrant de poids minimum.

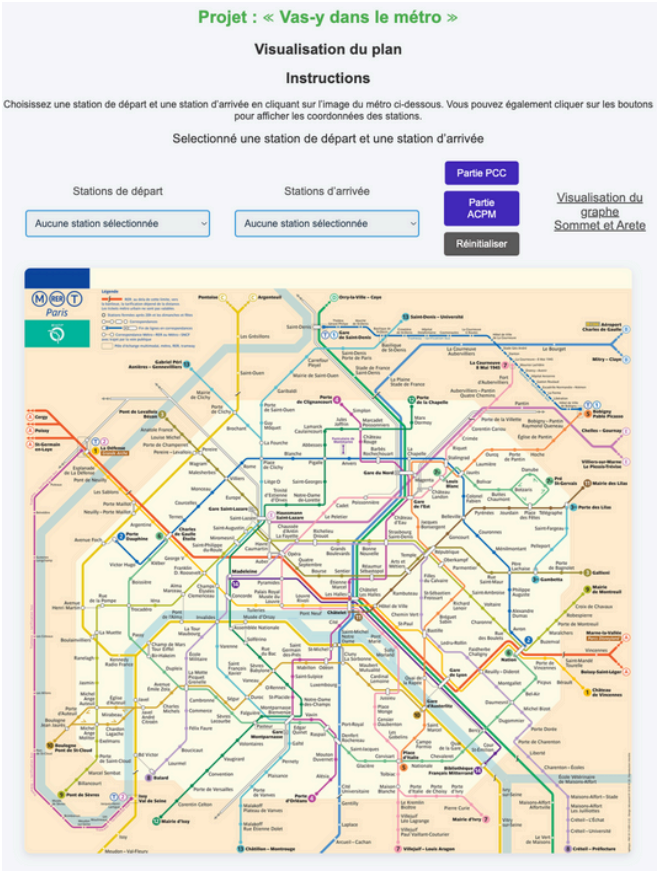
```python
def prim(graph: Graphe, random_sommet: Sommet = None) :
    if random_sommet is None:
        random_sommet = random.choice(graph.sommets)

    visited = set()
    arete_to_visite = set()
    acpm = []

    visited.add(random_sommet)
    current_aretes = graph.get_aretes_adjacentes(random_sommet)

    arete_to_visite.update(current_aretes)

    while len(visited) < len(graph.sommets):
        smallest_weighted_arete = smallest_weight(arete_to_visite)

        acpm.append(smallest_weighted_arete)

        visited.add(smallest_weighted_arete.sommet1)
        visited.add(smallest_weighted_arete.sommet2)

        for arete in graph.get_aretes_adjacentes(smallest_weighted_arete.sommet1):
            if arete.sommet1 not in visited or arete.sommet2 not in visited:
                arete_to_visite.add(arete)

        for arete in graph.get_aretes_adjacentes(smallest_weighted_arete.sommet2):
            if arete.sommet1 not in visited or arete.sommet2 not in visited:
                arete_to_visite.add(arete)
        arete_to_visite.remove(smallest_weighted_arete)

    return acpm
```

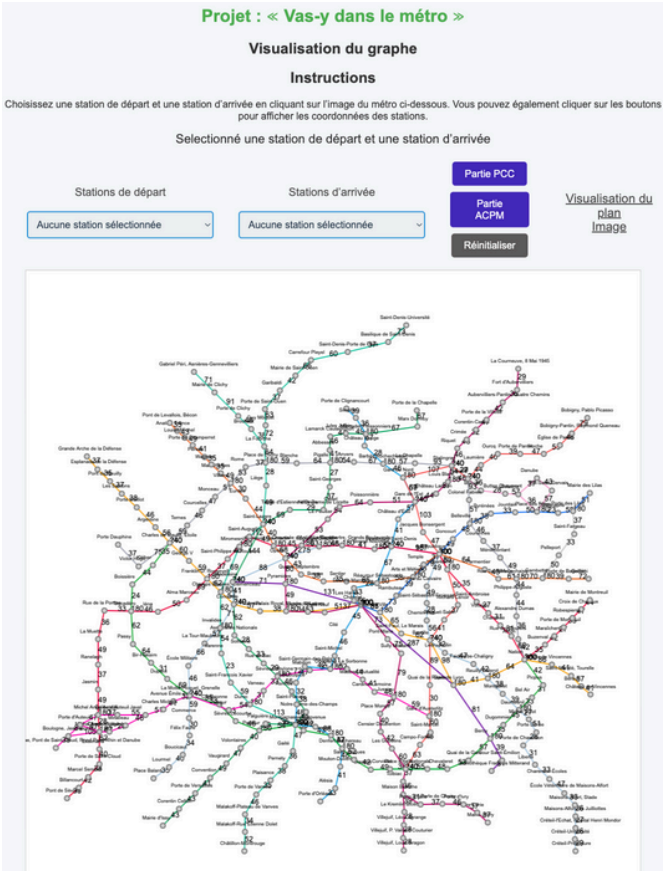
L'ACPM est utilisé pour déterminer un ensemble minimal de connexions entre les stations qui permet toujours un accès complet au réseau.

Pour cette algorithme de Prim on construit un arbre couvrant de poids minimal (ACPM) dans un graphe pondéré.
Il démarre depuis un sommet donné ou choisi aléatoirement,
On ajoute ajoutant progressivement l'arête de poids minimal qui connecte un sommet visité à un sommet non visité. Les sommets visités sont suivis dans un ensemble, et les arêtes disponibles sont mises à jour au fur et à mesure.
Une fois tous les sommets inclus, la fonction retourne les arêtes formant l'ACPM.

Bonus

Pour le bonus, nous avons développé des fonctionnalités en utilisant Flask afin de créer l'interface.

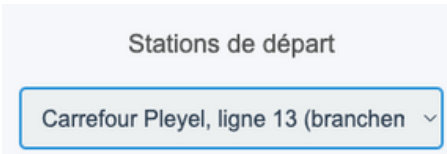


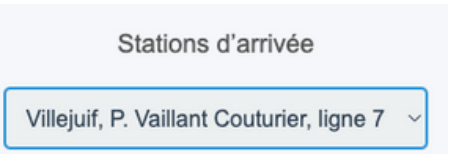


Fonctionnalité

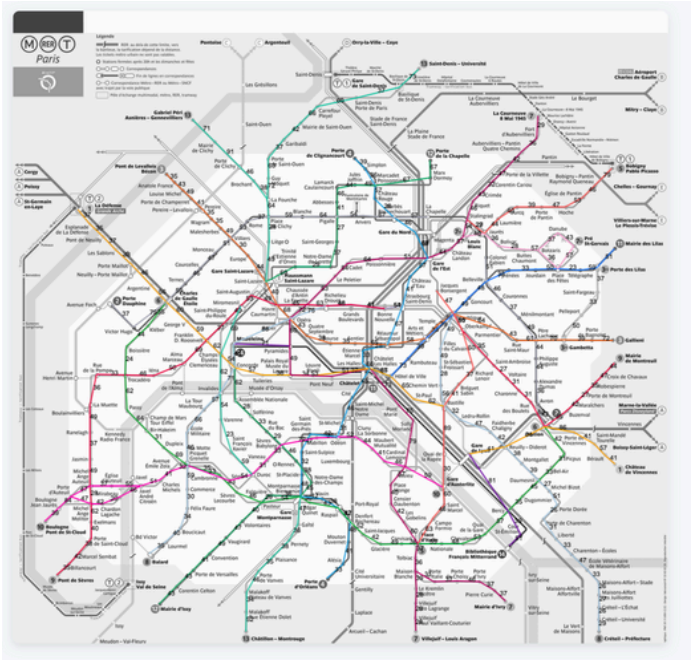
Pour la fonctionnalité du plus court chemin, on va utiliser les boutons de sélection pour mettre notre station de départ et d'arrivée. Après cela, on a simplement à appuyer sur PCC (pour le plus court chemin) pour voir le chemin ce dessiner.

On a aussi sur interface les stations et le temps





Pour la fonctionnalité de l'arbre couvrant de poids minimal, il est important de garder à l'esprit que l'ACPM reste constant, quel que soit le sommet de départ, en raison de la nature non orientée et connexe du graphe. Bien qu'il n'y ait pas d'animation pour visualiser la construction de l'ACPM, un bouton permet d'afficher l'ACPM sur la visualisation. On peut remarquer que certaines arêtes apparaissent en gris, car elles ne font pas partie de l'ACPM.



Conlusion

L'implémentation illustre une application réussie de la théorie des graphes dans un contexte pratique. Les algorithmes et les structures de données sont judicieusement exploités pour satisfaire les exigences du projet.
Parmi les améliorations envisageables, on pourrait optimiser l'affichage graphique, par exemple en ajoutant des animations pour l'acpm ou en veillant à ce que les traits suivent fidèlement les lignes sur la carte.
La qualité du code n'est pas optimal aussi on à une structure avancé mais peut grandement être simplifier ce qui va réduire la temps de nos algorithme.