Projet

Maths pour l'informatique

2024 - 2025

Maeva RAMAHATAFANDRY et Romain REN

15 novembre 2024

Table des matières

0 Contexte **2**

1 Objectif du projet **2**

2 Structures de données utilisées **3**
2.1 Structure de fichiers 3
2.2 Structure du graphe 3

3 Algorithmes implémentés **5**
3.1 Vérification de la connexité 5
3.2 Algorithme de plus court chemin 5
3.3 Arbre couvrant de poids minimum 7

4 Bonus **8**
4.1 PPC 8
4.2 ACPM 9

0 Contexte

Ce projet a été réalisé dans le cadre de la matière Maths pour l'informatique à Efrei Paris en 2024.

L'objectif étant de mettre en pratique les connaissances en théorie des graphes pour résoudre des problématiques liées au réseau de métro parisien.

Le projet a été codé en Python, un langage adapté pour manipuler des structures de données complexes et implémenter des algorithmes efficaces.

Ce projet a été réalisé par Maeva RAMAHATAFANDRY et Romain REN, étudiant ingénieur de la classe INGÉ1-APP.

Vous pouvez exécuter le code avec le script bash setup_and_run.sh

Veillez bien à avoir Python3 sur votre machine, l'installation des librairies avec le script nécessite pip qui est une librairie par défaut sur Python à partir de la version 3.4

1 Objectif du projet

Le projet consiste à modéliser le réseau de métro parisien comme un graphe non orienté $G(V, E)$, où les sommets V représentent les stations et les arêtes E représentent les connexions entre ces stations.

Nous avons implémenté plusieurs fonctionnalités majeures :

— Vérification de la **connexité** du graphe
— Calcul du plus court chemin entre deux stations utilisant l'algorithme de **Bellman-Ford**
— Construction d'un **arbre couvrant de poids minimum** (ACPM) avec l'**algorithme de Prim**.
— Une **interface graphique** permettant à l'utilisateur de **visualiser** et **interagir** avec la carte du métro pour **calculer des itinéraires**.

2 Structures de données utilisées

2.1 Structure de fichiers

Le projet est constitué de 3 fichiers :

— **main.py** : fichier principal où sont exécutées les fonctions ainsi que l'interface graphique
— **algo.py** : fichier annexe contenant les définitions des **fonctions des différents algorithmes**
— **test.py** : fichier annexe de test contenant **uniquement les tests** des fonctions sans l'interface graphique

Les fichiers sources du projet sont stockés dans un dossier project_file.

2.2 Structure du graphe

Le graphe a été représenté en **Python** sous forme de dictionnaire, où chaque clé est un sommet (station) et chaque valeur est une liste de tuples représentant les voisins et le poids (temps de trajet) de l'arête :

```python
def getGraphe(file):
    # Initialisation des listes contenant les stations et les arcs
    stations =  []
    aretes = []

    # Initialisation des expressions régulières utilisées pour
    récupérer les valeurs dans metro.txt
    vertex_regex = r"V (\d{4}) (.*?);(\d+|[0-9bis]+) ;(True|False) (\d)"
    edge_regex = r"E (\d+) (\d+) (\d+)"

    # Récupération des valeurs et
    ajout dans les listes stations et aretes
    for line in file:
        vertices = re.findall(vertex_regex, line)
        for v in vertices:
            stations.append([int(v[0]), # n° de station
            v[1][:-1], # nom
            v[2], # ligne de métro
            v[3], # terminus
            int(v[4])]) # n° de branchement

        edges = re.findall(edge_regex, line)
        for e in edges:
            aretes.append([int(e[0]), int(e[1]), int(e[2])])

    # Initialisation d'un dictionnaire graphe
    contenant les stations reliées entre elles
    graphe = {}
    for i in range(len(aretes)):
        if aretes[i][0] not in graphe:
            graphe[aretes[i][0]] = [(aretes[i][1], aretes[i][2])]
        else :
            graphe[aretes[i][0]].append((aretes[i][1], aretes[i][2]))
        if aretes[i][1] not in graphe:
            graphe[aretes[i][1]] = [(aretes[i][0], aretes[i][2])]
        else :
            graphe[aretes[i][1]].append((aretes[i][0], aretes[i][2]))

    return graphe, stations
```

Pour extraire et ordonner les informations du fichier *metro.txt*, nous avons utilisés la librairie **re**, une librairie permettant d'utiliser les **expressions régulières** qui permettent de rechercher ou d'extraire dans des lignes de textes des informations de manières plus précises et flexibles à l'aide de **modèles de recherche**.

3 Algorithmes implémentés

3.1 Vérification de la connexité

Pour la fonction de vérification de connexité prenant en paramètre le graphe des stations et retournant **True** ou **False** selon si le graphe est connexe ou non, nous avons utilisé un **parcours en largeur** avec une **file des stations à visiter** et une **liste des stations déjà visités**.

```
def isConnexe(graphe):
    debut = next(iter(graphe)) # choix d'un sommet arbitraire
    file = [debut] # file des éléments à visiter
    parcours = [debut] # liste des sommets visités

    while file:
        noeud = file.pop(0) # en enlève l'élément en tête de file

        # parcours des voisins
        for v, _ in graphe[noeud]:
            if v not in parcours: # si on n'a pas encore visité le sommet
                parcours.append(v)
                file.append(v)
    return len(parcours) == len(graphe)
```

3.2 Algorithme de plus court chemin

Pour la fonction déterminant l'algorithme de plus court chemin nous avons utilisées l'**algorithme de Bellman-Ford**.

Cet algorithme est capable de gérer les graphes avec des poids négatifs, mais dans le cadre de ce projet, les poids représentent des **durées positives de trajet**.

```python
def bellman_ford(graph, station_debut, station_fin):
    debut = None
    fin = None
    for i in range(len(stations)):
        if (stations[i][1] == station_debut):
            debut = stations[i][0]
        elif (stations[i][1] == station_fin):
            fin = stations[i][0]

    if (debut == None or fin == None):
        print("Erreur : Nom de station inexistant")
        return None

    # Initialise les distances
    distances = {noeud: float('inf') for noeud in graph}
    distances[debut] = 0
    predecesseurs = {noeud: None for noeud in graph}

    for i in range(len(graph) - 1):
        for noeud in graph:
            for voisin, poids in graph[noeud]:
                if distances[noeud] + poids < distances[voisin]:
                    distances[voisin] = distances[noeud] + poids
                    predecesseurs[voisin] = noeud

    # Construit le chemin le plus court
    chemin = []
    station_actuel = fin
    while station_actuel is not None:
        chemin.insert(0, station_actuel)
        station_actuel = predecesseurs[station_actuel]

    station_precedente = None
    print("Le chemin le plus court est : ")
    for i in range(len(chemin)):
        if (stations[chemin[i]][1] == station_precedente):
            print("Changement de ligne " + stations[chemin[i]][2]
                                         + " : " +stations[chemin[i]][1])
        else :
            print(stations[chemin[i]][2] + " : " +stations[chemin[i]][1])
        station_precedente = stations[chemin[i]][1]
    print("La distance totale est : ", distances[fin]//60,
          " minutes et" , distances[fin]%60, "secondes.")

    return chemin, distances[fin]
```

3.3 Arbre couvrant de poids minimum

Pour obtenir l'arbre couvrant de poids minimum, nous avons utilisé l'algorithme de Prim, qui construit progressivement l'ACPM en choisissant d'abord les arêtes de plus petits poids.

La fonction prend en paramètre le graphe, et renvoie à la fin une liste similaire au graphe mais correspondant à l'arbre couvrant de poids minimum.

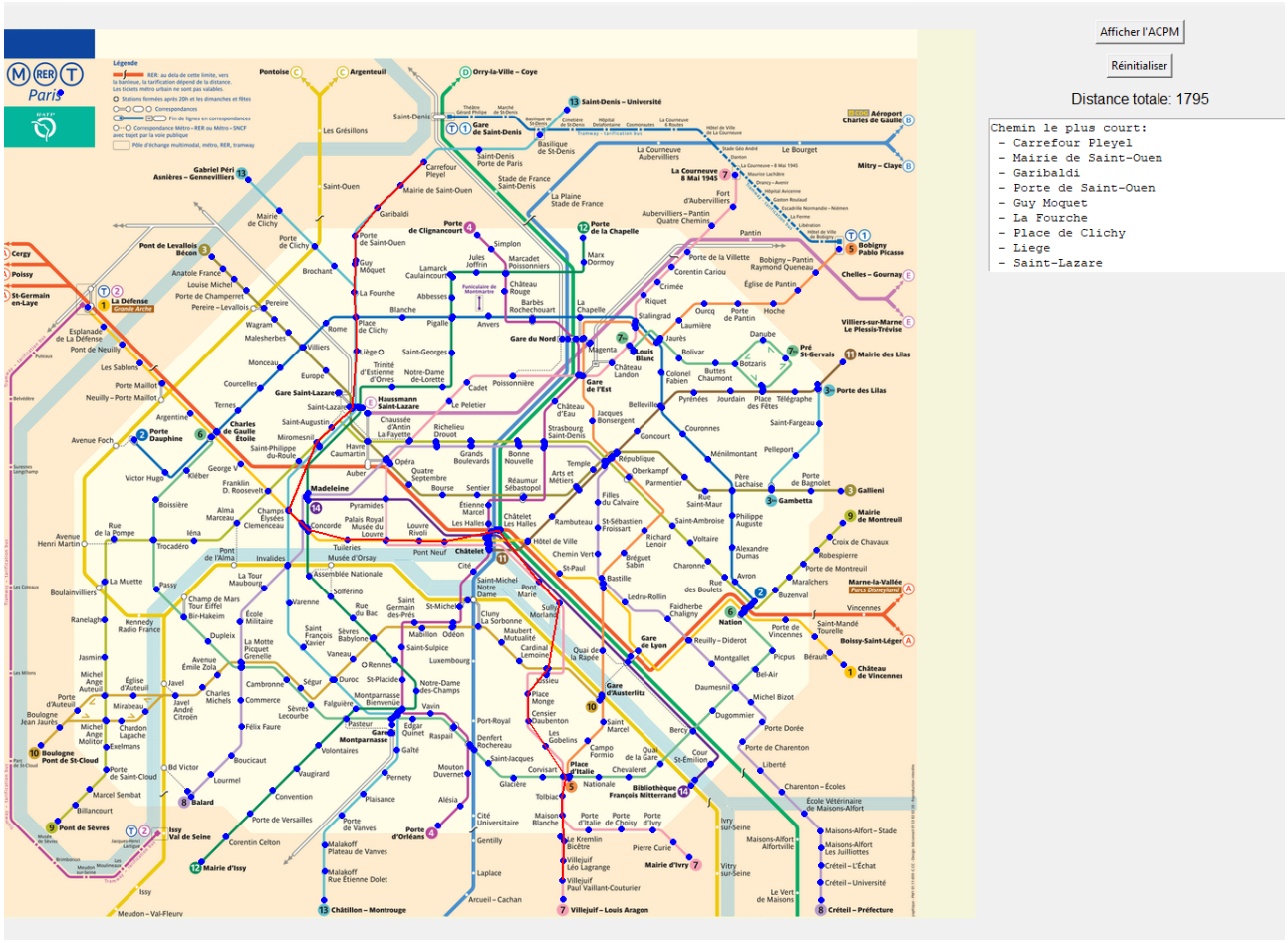
```
def arbreCouvrant(graphe):
    debut = next(iter(graphe)) # choix arbitraire du départ

    # initialisations
    arbre = [] # liste des arêtes de l'arbre couvrant minimal
    visites = set([debut])
    aretes = [(poids, debut, voisin) for voisin, poids in graphe[debut]]

    while aretes:
        aretes.sort() # on trie d'abord les arêtes dans l'ordre croissant
        poids, u, v = aretes.pop(0) # on choisit l'arete de plus petit poids

        if v not in visites: # si le sommet n'a pas encore été visité,
                             on l'ajoute dans l'arbre
            arbre.append((u, v, poids))
            visites.add(v)

            for voisin, poids_voisin in graphe[v]:
            # on rajoute les nouveaux sommets à vérifier
            (voisin du sommet visité )
                if voisin not in visites:
                    aretes.append((poids_voisin, v, voisin))
    return arbre
```

4 Bonus

Pour la partie **Bonus**, nous avons utilisé **Tkinter**, nous avons affiché la carte du métro avec des **points clickables** par dessus pour la carte pour indiquer les stations de métro.

4.1 PPC

Lorsque vous cliquez sur 2 stations (les points bleus) sur la carte, une route (la ligne rouge) est affichée indiquant le plus court chemin. La liste est affichée sur l'interface dans une **boite de texte à droite** de la carte.



4.2 ACPM

Concernant l'ACPM, un bouton **Afficher l'ACPM** est affichée à droite en cliquant dessus, l'ACPM avec des lignes rouges s'affichent sur la carte.