TD 29 : Graphes : Dijkstra

Récupérer le fichier `metro_paris.txt` de 1311 lignes, au format utf-8, qui contient la description du graphe orienté du métro parisien, de manière naïve :

- une ligne d'entête ([Vertices])
- une liste de 376 stations (sommets) : chaque ligne du fichier est constituée d'un numéro de station, d'une espace, du nom de la station, d'un caractère de fin de ligne.
- une ligne d'entête ([Edges])
- une liste de 933 arcs du graphe, c'est-à dire des couples de stations voisines, reliées par une ligne de métro ou par une correspondance. Ainsi lorsqu'une station appartient à plusieurs lignes de métro, elle apparaît plusieurs fois. Chaque ligne du fichier est constituée du numéro d'une station, d'une espace, du numéro d'une autre station, d'une espace, du temps en seconde pour aller de la première à la seconde station, d'un caractère de fin de ligne. Les correspondances sont placées à la fin, avec un temps conventionnel de deux minutes. **On ne fait aucune hypothèse sur l'ordre d'apparition des arcs dans le fichier.**

L'objectif du TD est de réinvestir les activités des TD précédents : lecture du fichier texte, extraction des données du graphe, parcours en largeur. Mise en application de l'algorithme de Dijkstra pour rechercher un plus court chemin dans un graphe pondéré réel.

Exercice 1 (Travail préliminaire : liste d'adjacence du graphe pondéré)

On s'intéresse au **graphe orienté pondéré** du métro parisien décrit dans le fichier `metro_paris.txt`.

1. **Lecture du fichier texte**

   Récupérer les informations du fichier texte : créer une représentation naïve du graphe orienté sous la forme basique `G = (S,A)`, où S est l'ensemble des **noms** des sommets (stations) et A l'ensemble des **arcs valués**, sous la forme de triplets (s,s',w). Les sommets s et s' sont des entiers et le poids w de l'arc liant s à s' est un flottant.

2. **Création de la représentation par liste d'adjacences pondérée**

   Écrire une fonction `liste_adjacence(G)` qui prend en argument un graphe $G = (S, A)$ naïf (du type de celui qui vient d'être créé à la question précédente) et qui retourne un tableau (liste Python) contenant la représentation du graphe G par liste d'adjacence. Appliquer cette fonction au graphe obtenu à la question précédente pour définir la variable globale `graphe_metro`. On doit obtenir une représentation par liste d'adjacence du type :

   [[(238, 41.0), (159, 46.0)], [(12, 36.0), (235, 44.0)], [(110, 69.0), (139, 50.0)], [(262, 33.0), (210, 41.0)],$\cdots$]

   Le premier voisinage [[(238, 41.0), (159, 46.0)] indique avec le couple (238, 41.0) que la station Abbesses (sommet 0) est reliée à la station Pigalle (238) en 41 secondes. Consulter le plan du métro fourni dans le fichier `carte-metro-paris.pdf` (cette carte contient davantage de stations que le graphe et montre aussi les lignes de RER).

Exercice 2 (Plus court chemin dans un graphe pondéré orienté, algorithme de Dijkstra)

On utilise le module `queue` et la structure de file de priorité `PriorityQueue` qu'il contient. Une file de priorité vide est créée par `F = PriorityQueue()`, on enfile par `F.put((p,s))`, on défile par `F.get()` et on teste par `F.empty()`.

1. **Algorithme de Dijkstra**

   Rappeler l'algorithme de Dijkstra.

   Écrire une fonction `Dijkstra(g,s)` qui retourne le tableau des distances (poids total des chemins) et le tableau des prédécesseurs pour les plus courts chemins trouvés.

2. **Reconstitution du plus court chemin**

   Écrire une fonction `plus_court_chemin(g,s1,s2)` qui retourne le plus court chemin de `s1` à `s2` donné par la fonction Dijkstra précédente, et le poids de ce chemin, sous forme d'une liste de sommets, de `s1` à `s2` et d'un flottant.

3. Tester la fonction `plus_court_chemin(g,s1,s2)` sur le graphe du métro. Par exemple pour aller de la station Avron à la station Victor Hugo, qui sont sur la même ligne 2, il est avantageux de changer à Nation et à Charles de Gaulle, Étoile.

4. Adapter la fonction `bfs(g,s)` écrite dans un TD précédent pour réaliser un simple parcours en largeur et trouver le plus court chemin sans tenir compte des pondérations (c'est-à-dire en nombre d'étapes). Pour implémenter la file, utiliser la structure `Queue` du module `queue` avec les mêmes instructions. Pour tester, on pourra récupérer les variables obtenues et sauvegardées dans un fichier `metro_paris_liste_adjacence` lors d'un précédent TD.

5. - Compléter les fonctions pour afficher les temps de parcours dans les différents cas.

   - Comparer différents plus courts chemin proposés par l'algorithme de Dijkstra et ceux proposés par un parcours en largeur qui ne tient pas compte des temps de parcours.

   - Par exemple pour aller de la station Avron à la station Victor Hugo, le plus court chemin en nombre de stations contient 4 changements (Nation, Gare de Lyon, Châtelet, Charles de Gaulle, Étoile), il est plus long en temps (23,4 minutes au lieu de 19,8 minutes pour le parcours de Dijkstra).

   - Reprendre la recherche du plus court chemin de Créteil-Préfecture à Pont de Sèvres avec l'algorithme de Dijkstra et constater que l'on gagne plus de 7 minutes en moyenne sur le plus court chemin du parcours en largeur.

Ce qu'il faut retenir

L'informatique de tronc commun est évaluée à l'écrit des concours d'entrée dans les grandes écoles. Il est donc avant tout nécessaire de connaître les algorithmes, les structures de données utilisées et surtout d'être capable de dérouler « à la main » les différents algorithmes.

Appliquer l'algorithme de Dijkstra « à la main » pour déterminer les plus courts chemins, et leurs poids, depuis le sommet 0 : préciser l'état de la file de priorité à chaque étape.