

GRAPHES - PROJET MÉTRO PARISIEN

Ces exercices sont adaptés d'après *Informatique pour tous A. Caïgnot 2015 Vuibert Prépas.*

Le but de ces exercices est de fournir un algorithme capable de donner le meilleur itinéraire (en termes de temps de parcours) pour relier deux stations du métro parisien.

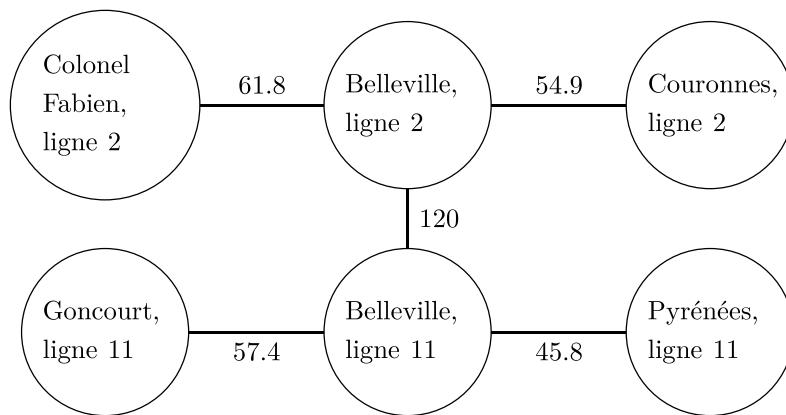
Exercice 1 (Métro parisien) Pour simplifier le problème, nous allons faire les restrictions suivantes :

- seules les stations de métro sont prises en compte (pas de RER, pas de bus ...);
- toutes les rames sont supposées se déplacer à la même vitesse;
- les seules correspondances possibles sont celles d'une même station sur deux lignes différentes (pas de correspondance à pied du type La Chapelle-Gare du Nord);
- Toutes les correspondances prennent le même temps : 120 secondes (même si, en réalité, certaines correspondances, comme celle de Montparnasse, prennent plus de temps).

Le métro est représenté par un graphe étiqueté non orienté tel que :

- un sommet du graphe correspond à une station sur une ligne donnée;
- il existe une arête entre deux sommets si les deux stations sont à la suite de l'autre sur une ligne ou s'il existe une correspondance à cette station entre deux lignes;
- le poids d'une arête est égal au temps de trajet entre les deux sommets.

Par exemple, la portion du graphe du métro au niveau de la station Belleville est donné ci-dessous.



Comme le graphe du métro est peu dense (un sommet est relié à un nombre peu élevé d'autres sommets), une matrice d'adjacence n'est pas un choix judicieux pour représenter ce graphe car elle serait essentiellement remplie de $+\infty$.

Nous utiliserons une représentation par liste d'adjacence, c'est-à-dire par une liste `g` telle que `g[i]` contient une liste de tuples (`voisin, poids`) avec tous les voisins de `i`.

Il faut encore pouvoir identifier chaque tuple (`station, ligne`) de manière unique. Pour ce faire, on va utiliser un dictionnaire dont la clé est le tuple (`station, ligne`), `station` et `ligne` étant de type `str` et la valeur, un nombre entier unique appelé `index`.

Voici les quelques lignes de code permettant de récupérer les données qui construisent le graphe du métro à partir du fichier `metro.pkl` (à mettre dans le même répertoire que le fichier python).

```
import pickle  
fich = open('metro.pkl','rb')  
graphMetroTest = pickle.load(fich)  
graphMetro = pickle.load(fich)  
dicMetro = pickle.load(fich)  
fich.close()
```

Le module `pickle` permet de stocker et de restaurer dans un fichier, au format binaire, n'importe quel objet Python comme une liste, un dictionnaire, un tuple ... sans devoir utiliser une base de données ou un fichier texte.

Deux graphes sont disponibles :

- `graphMetro` qui représente le métro parisien complet ;
- `graphMetroTest` qui est restreint aux lignes 1 et 2, et prévu pour tester les fonctions.

Ces deux graphes sont définis en tant que variables globales. Elles ne seront jamais modifiées par les fonctions, juste lues.

- 1) Soit `n` le nombre de sommets d'un graphe et `a` le nombre de ses arêtes.
 - a) En termes de complexité spatiale, comparer la représentation par matrice d'adjacence à la représentation par liste d'adjacence.
 - b) Donner la valeur limite de `a` en-dessous de laquelle la représentation par liste d'adjacence est plus économique.
- 2) Déterminer le nombre de sommets et le nombre d'arêtes de `graphMetro` et de `graphMetroTest`.
- 3) Ecrire la fonction `indexSommet` qui, à partir des paramètres `station` et `ligne`, tous deux de type `str`, permet de récupérer le numéro unique de type entier.
- 4) Ecrire la fonction `infosStation` qui, à partir du paramètre `indexSommet`, de type entier, retourne un tuple (`station`, `ligne`), `station` et `ligne` étant de type `str`.
Utiliser cette fonction pour connaître le numéro de la station Belleville sur la ligne 2.
- 5) Ecrire une fonction `voisins` qui prend en entrée le numéro de sommet `indexSommet` et renvoie la liste des voisins de ce sommet, plus précisément une liste de tuples (`indexVoisin`, `poids`) avec tous les voisins de `indexSommet`.
Utiliser cette fonction pour connaître les voisins de la station Belleville sur la ligne 2.
- 6) Ecrire une fonction `existe` qui prend en entrée deux numéros de sommets et renvoie `True` s'il existe une arête entre ces deux sommets, et `False` sinon.
Utiliser cette fonction pour vérifier s'il existe une arête ou pas entre la station Belleville sur la ligne 2 et
 - la station Couronnes sur la ligne 2 ;
 - la station Belleville sur la ligne 11 ;
 - la station Goncourt sur la ligne 11.
- 7) Ecrire une fonction `poids` qui prend en entrée deux numéros de sommets et renvoie la distance entre ces deux sommets s'il existe une arête entre ces deux sommets, et -1 sinon.
Dans le contexte de l'exercice, la `distance` est le temps de trajet entre deux stations.
Reprendre les stations et les lignes de la question précédentes pour déterminer les distances entre ces stations.
- 8) Ecrire une fonction `arêtes` qui renvoie la liste des arêtes du graphe sous forme de tuples (une arête entre les sommets de numéros `i` et `j` est représentée par le tuple (`i`, `j`)).

Exercice 2 (Métro - Recherche du plus court chemin)

L'algorithme de Dijkstra est un algorithme de recherche du plus court chemin.

E. W. Dijkstra (mathématicien et informaticien néerlandais) a proposé en 1959 un algorithme de recherche du plus court chemin.

Il s'agit d'un algorithme glouton : lorsque plusieurs choix sont possibles, l'algorithme glouton fait le choix localement optimal dans l'espoir d'obtenir au final une solution globalement optimale. Les algorithmes gloutons ne donnent pas toujours une solution optimale, mais, dans la recherche du plus court chemin, la solution obtenue est optimale.

Cet algorithme ne fonctionne que si le graphe ne possède que des arêtes dont les étiquettes sont à valeurs positives. C'est l'un des plus efficaces pour traiter les problèmes de plus court chemin. Grâce à la puissance du traitement informatique, il est utilisé par les logiciels d'optimisation de trajets réels (navigateurs GPS, site RATP...) ou virtuels (routage internet).

L'algorithme de Dijkstra est basé sur le principe suivant : si le plus court chemin reliant le sommet `dep` (sommet de départ) au sommet `s` (sommet final) passe par les sommets s_1, s_2, \dots, s_k , alors les différentes étapes sont aussi les plus courts chemins reliant `dep` aux sommets successifs s_1, s_2, \dots, s_k .

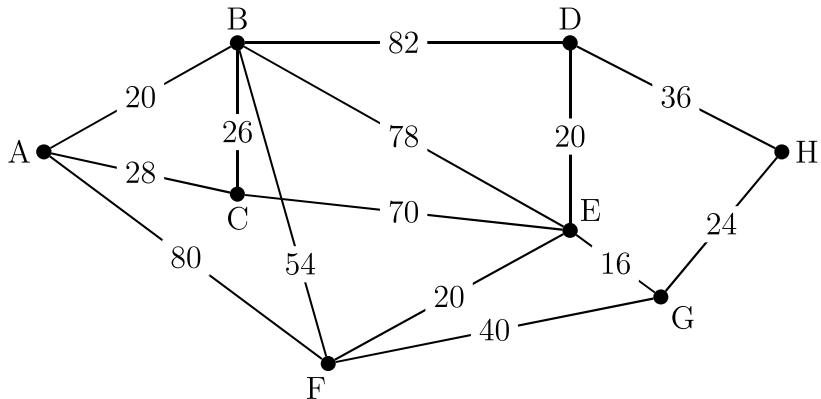
Il s'agit donc construire de proche en proche le chemin cherché en choisissant à chaque itération de l'algorithme, un sommet s_i du graphe parmi ceux qui n'ont pas encore été traités, tel que la longueur connue provisoirement du plus court chemin allant de `dep` à s_i soit minimale.

Le déroulement de l'algorithme de Dijkstra est le suivant.

- Initialisation de 2 listes
 - Une liste de sommets non visités, nommée `nonVisites`, est initialisée avec la liste des sommets du graphe.
 - Une liste `distances` comprenant les distances `d(dep, s)` entre `dep` et chaque sommet `s` du graphe est initialisée avec des $+\infty$ (`float('inf')` en Python). La valeur correspondant au sommet de départ est fixée à zéro.
 - Tant que la liste des sommets non visités est non vide (c'est-à-dire tant que tous les sommets n'ont pas été visités) :
 - le sommet non visité dont la distance est minimale est noté `s` et est enlevé de la liste des sommets non visités
 - pour tous les voisins `v` de `s`, on stocke dans la liste `distances` leur distance minimale au sommet de départ (en prenant le minimum entre `d(dep, v)` (leur distance actuelle dans la liste) et celle obtenue en additionnant `d(dep, s)+d(s, v)` (la nouvelle distance obtenue en ajoutant le chemin de `s` à `v`)).
- 1) Ecrire une fonction `enleve` qui prend en entrée une liste et un élément `elt`, et renvoie la liste privée de cet élément. La liste initiale contient au plus une fois l'élément à enlever.
 - 2) Ecrire une fonction `dijkstra` qui prend en entrées, un graphe et un sommet de départ, et renvoie la liste des distances entre chaque sommet et le sommet de départ.
 - 3) Quelle est la complexité temporelle de cet algorithme ? L'unité de coût choisie est la comparaison de deux éléments.
 - 4) Améliorer la fonction précédente pour qu'elle renvoie aussi une liste des prédécesseurs permettant de retrouver le chemin de distance minimale.
On appellera cette fonction `dijkstraChem`.
 - 5) Ecrire une fonction `cheminD` qui prend en entrées, un graphe, un sommet de départ et un sommet d'arrivée, et renvoie la liste des identifiants des stations du chemin minimal dans l'ordre dans lequel elles sont parcourues.
 - 6) Ecrire une fonction `afficheChemin` qui prend en entrée une liste de numéros de stations et affiche la description détaillée en français de l'itinéraire à suivre (Monter à telle station, telle ligne, donner les stations suivantes, indiquer les correspondances, indiquer où descendre ...).

Complément - Algorithme de Dijkstra "à la main" sur un exemple

Le réseau ferroviaire d'une région est schématisé par le graphe ci-dessous. Les sommets représentent les villes et les arêtes représentent les voies ferrées. Sur les arêtes du graphe sont indiquées les distances exprimées en kilomètre entre les villes de la région.



Déterminer, en utilisant l'algorithme de Dijkstra, le trajet le plus court pour aller de la ville A à la ville H. Préciser la longueur en kilomètre de ce trajet.

A	B	C	D	E	F	G	H	On enlève
0	∞	∞	∞	∞	∞	∞	∞	A
20 A	28 A	∞	∞	∞	80 A	∞	∞	B
	28 A	∞	102 B	98 B	80 A 74 B	∞	∞	C
		46 B	102 B	98 B 98 C	74 B	∞	∞	F
			102 B	98 B 94 F		114 F	∞	E
			102 B	102 B 94 F		114 F 110 E	∞	D
						110 E	138 D	G
							138 D 134 G	H

Le trajet le plus court est : A $\xrightarrow{20}$ B $\xrightarrow{54}$ F $\xrightarrow{20}$ E $\xrightarrow{16}$ G $\xrightarrow{24}$ H ; il a une longueur de 134 km.

Code pour tester :

```

graphemeMini = [[(1,20),(2,28),(5,80)],
[(0,20), (2,26),(3,82),(4,78),(5,54)],
[(0,28),(1,26),(4,70)],[(1,82), (4,20),(7,36)],
[(1,78), (2,70),(3,20),(5,20),(6,16)],
[(0,80),(1,54), (4,20),(6,40)],[(4,16),(5,40), (7,24)],[(3,36), (6,24)]]

print(dijkstra(graphemeMini,0)) # [0, 20, 28, 102, 94, 74, 110, 134]

distances, pred = dijkstraChem(graphemeMini,0)
print(distances) # [0, 20, 28, 102, 94, 74, 110, 134]
print(pred) # [-1, 0, 0, 1, 5, 1, 4, 6]
  
```