Plus courts chemins dans un graphe pondéré

L'algorithme de Dijkstra

Marc Lorenzi - Février 2018

In [1]:

```
import prioqueue
import utils
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
%matplotlib inline
```

- Le module `prioqueue` implémente la structure de file de priorité à partir de tas min. Le code se trouve dans le fichier `prioqueue.py`, dans le même répertoire que ce notebook.
- Nous reparlerons du module `utils` dans la section 4.

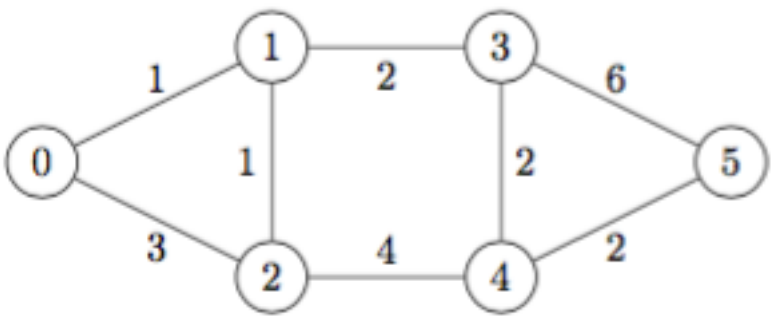
1. Introduction

Soit $G = (S, A, f)$ un graphe pondéré. S est l'ensemble des sommets de G, A est l'ensemble des arêtes, et $f : A \to \mathbb{R}_+$ est une fonction associant un poids à chaque arête du graphe. L'algorithme de Dijkstra permet de calculer les plus courts chemins entre un sommet de G et tous les autres sommets de sa composante connexe.

Sans restreindre la généralité, on suppose que $S = \{0, 1, \dots, n-1\}$ où n est le nombre de sommets de G. On représente le graphe G en Python par une liste de taille n. Pour $i = 0, \dots, n-1$, le $i$ème élément de la liste est la liste des couples (j, p) où j décrit l'ensemble des voisins de i et p est le poids de l'arête $\{i, j\}$. Pour les tests nous prendrons les deux exemples ci-dessous, qui sont tous les deux connexes et non orientés.

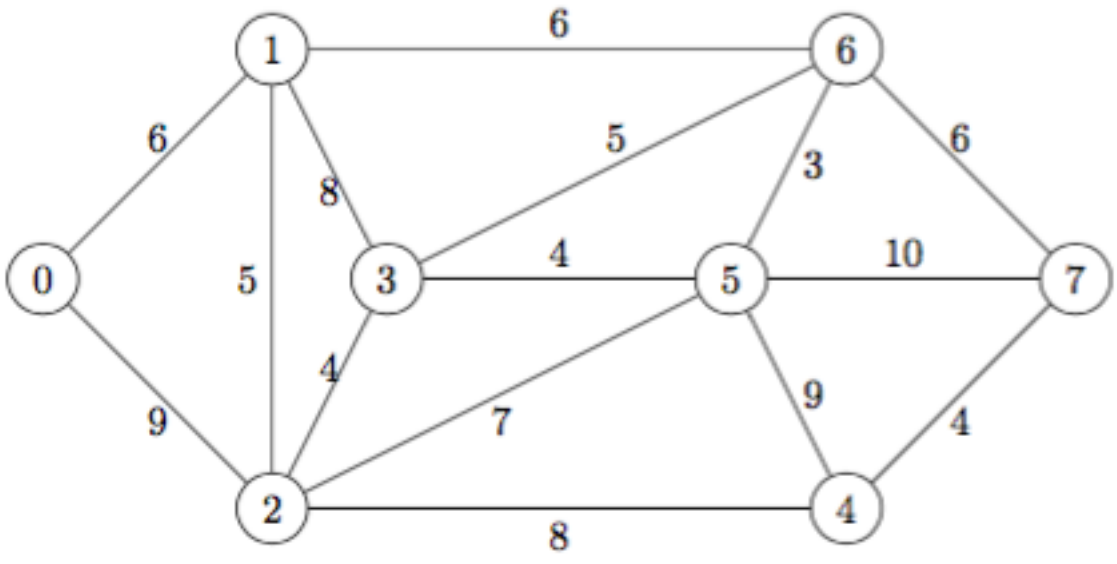
In [2]:

```
ex01 = [[(1,1),(2,3)],[(0,1),(2,1),(3,2)],[(0,3),(1,1),(4,4)],
  [(1,2),(4,2),(5,6)],[(2,4),(3,2),(5,2)],[(3,6),(4,2)]]
```



In [3]:

```
ex02 = [[(1,6),(2,9)],[(0,6),(2,5),(3,8),(6,6)],[(0,9),(1,5),(3,4),(4,8),(5,7)
],
  [(1,8),(2,4),(5,4),(6,5)],[(2,8),(5,9),(7,4)],[(2,7),(3,4),(4,9),(6,3),(7,10
)],
  [(1,6),(3,5),(5,3),(7,6)],[(4,4),(5,10),(6,6)]]
```



Les algorithmes de parcours de graphe nécessitent de marquer les sommets lors de leur visite. Nous définissons donc les deux couleurs `Blanc` et `Noir` correspondant respectivement aux sommets marqués et non marqués.

In [4]:

```
Blanc = 0
Noir = 1
```

2. L'algorithme de Dijkstra

La fonction `dijkstra` prend en paramètres un graphe g et un sommet "racine" r. Elle renvoie un couple (peres, distances) où `peres[j]` est le père de j dans un arbre couvrant enraciné en r et `distances[j]` est la distance de r à j dans le graphe g.

Au départ tous les pères sont initialisés à -1 (pas de père) et les distances à -1 (distance "infinie" de r au sommet concerné). La fonction utilise une file de priorité. Lorsqu'un sommet est mis dans la file, sa priorité est égale à sa distance (provisoire) au sommet racine. Au départ, on met r dans la file avec la priorité 0. Tant que la file est non vide, on défile le sommet u dont la distance provisoire à r est minimale. On peut en fait montrer que cette distance est définitive et qu'on a trouvé le plus court chemin de r à u. On noircit u, puis on visite les voisins de u.

Une fois un sommet noirci, les informations relatives à ce sommet (père, distance à r) sont définitives. En revanche, un sommet blanc peut voir sa distance à r et son père réévalués lors de l'exécution de l'algorithme.

Pour un code plus propre, nous définissons une classe `Graphe`.

In [5]:

```
class Graphe:

    def __init__(self, aretes):
        self.aretes = aretes
        self.init_data()

    # Nombre de sommets
    def __len__(self):
        return len(self.aretes)

    # Initialisation des champs du graphe (hormis ses arêtes)
    def init_data(self):
        n = len(self.aretes)
        self.peres = n * [-1]
        self.dist = n * [-1]
        self.couleurs = n * [Blanc]
```

Pour créer un graphe il suffit de passer en paramètre au constructuer la liste de ses arêtes.

In [6]:

```
gex01 = Graphe(ex01)
gex02 = Graphe(ex02)
```

In [7]:

```
def dijkstra(G, r):
    f = prioqueue.Prioqueue()
    G.init_data()
    G.dist[r] = 0
    f.push(r, 0)
    while not (f.is_empty()):
        (u, p) = f.pop()
        G.couleurs[u] = Noir
        visiter_voisins(G, u, f)
    return (G.peres, G.dist)
```

Reste bien entendu à écrire la fonction `visiter_voisins`. Pour chacun des voisins blancs v de u, on calcule `dist[u]+g(u,v)`. Si cette quantité est inférieure à `dist[v]`, on vient de trouver un chemin de r à v plus court que celui que l'on connaissait auparavant. On modifie donc `peres` et `distances`en conséquence :

- u devient le père de v dans le tableau des pères.
- On ajuste la distance de v à la nouvelle valeur.
- On met v dans la file s'il n'y était pas et on diminue sa priorité s'il y était déjà.

In [8]:

```
def visiter_voisins(G, u, f):
    for (v, d) in G.aretes[u]:
        if G.couleurs[v] == Blanc and (G.dist[v] > G.dist[u] + d or G.dist[v]
== -1):
            G.peres[v] = u
            G.dist[v] = G.dist[u] + d
            if not f.is_in(v):
                f.push(v, G.dist[v])
            else:
                f.decrease_prio(v, G.dist[v])
```

Testons sur les deux exemples avec le sommet 0 pour racine.

In [9]:

```
dijkstra(gex01, 0)
```

Out[9]:

```
([-1, 0, 1, 1, 3, 4], [0, 1, 2, 3, 5, 7])
```

Ainsi, par exemple, la distance du sommet 4 au sommet 0 est 5, et un chemin ayant cette longueur est `4->3->1->0` car le père de 4 est 3, le père de 3 est 1 et le père de 1 est 0.

In [10]:

```
dijkstra(gex02, 0)
```

Out[10]:

```
([-1, 0, 0, 2, 2, 6, 1, 6], [0, 6, 9, 13, 17, 15, 12, 18])
```

Un chemin de longueur minimale 18 du sommet 7 vers le sommet 0 est donc `7->6->1->0`.

Remarquons que les tableaux de pères et de distances sont renvoyés par la fonction `dijkstra`, mais qu'ils sont également préservés dans la structure de graphe.

In [11]:

```
gex02.peres, gex02.dist
```

Out[11]:

```
([-1, 0, 0, 2, 2, 6, 1, 6], [0, 6, 9, 13, 17, 15, 12, 18])
```

3. Calculs de chemins

Les deux exemples nous l'ont montré : il est facile, grâce à un tableau de pères et à la connaissance du sommet racine, de calculer un chemin d'un sommet u quelconque au sommet racine à partir duquel le tableau de pères a été calculé.

La fonction `chemin` calcule un tel chemin. La fonction `tous_chemins` renvoie quant à elle la liste de tous les chemins des sommets du graphe vers le sommet racine.

In [12]:

```
def chemin(peres, u, r):
    c = []
    while u != -1 and u != r:
        c.append(u)
        u = peres[u]
    if u == r:
        c.append(r)
        return c
    else: raise Exception
```

In [13]:

```
p, d = dijkstra(gex02, 0)
chemin(p, 7, 0)
```

Out[13]:

```
[7, 6, 1, 0]
```

In [14]:

```
def tous_chemins(peres, r):
    n = len(peres)
    cs = []
    for i in range(n):
        if peres[i] != -1:
            cs.append(chemin(peres, i, r))
    return cs
```

In [15]:

```
tous_chemins(p, 0)
```

Out[15]:

```
[[1, 0], [2, 0], [3, 2, 0], [4, 2, 0], [5, 6, 1, 0], [6, 1, 0], [7
, 6, 1, 0]]
```

4. Un mini Google Maps

In [16]:

```
plt.rcParams["figure.figsize"] = [12, 12]
```

Confrontons-nous à un exemple plus intéressant. Le module `utils` contient des fonctions permettant d'afficher une carte de France avec quelques routes reliant des villes voisines (une par département, hormis les départements 92, 93 et 94).

- La fonction `lire_france` renvoie une structure de données g contenant toutes les informations (positions, distances) sur les villes et les routes. `Graphe(g.edges)` est un graphe au sens où nous l'entendons dans ce notebook.
- La fonction `voir_chemins` prend en paramètres la structure g et une liste de chemins et les affiche en rouge sur la carte.

NB : Les distances étant calculées à vol d'oiseau, les informations renvoyées par nos fonctions ne doivent évidemment pas être prises au pied de la lettre !

Voici une fonction qui prend en paramètres deux numéros de département et affiche le plus court chemin entre les villes correspondantes.

In [17]:

```
def test1(dept1, dept2):
    g = utils.lire_france()
    p, d = dijkstra(Graphe(g.edges), dept1)
    c = chemin(p, dept2, dept1)
    print(d[dept2])
    utils.voir_chemins(g, [c])
```

Cherchons le plus court chemin de Strasbourg à Bordeaux et renvoyons la distance entre ces deux villes.

In [18]:

```
test1(67, 33)
```

806.458341485329

Voici pour finir une fonction qui prend en paramètre un numéro de département et affiche en rouge les routes de la ville correspondante vers tous les autres départements.

In [19]:

```
def test2(dept):
    g = utils.lire_france()
    p, d = dijkstra(Graphe(g.edges), dept)
    c = tous_chemins(p, dept)
    utils.voir_chemins(g, c)
```

Et voici toutes les routes de Poitiers aux autres villes de France.

In [20]:

```
test2(86)
```

Juste pour rire, quelle serait la distance totale parcourue en faisant un aller-retour de Poitiers vers toutes les autres villes de France ?

In [21]:

```
g = utils.lire_france()
```

In [22]:

```
p, d = dijkstra(Graphe(g.edges), 86)
```

In [23]:

```
s = 0
for x in d:
    if x != -1:
        s = s + 2 * x
print(s)
```

```
64146.362190206615
```

Une fois et demi le tour de la Terre. À vol d'oiseau :-).