Université Clermont Auvergne
BUT Info 1A
R207 - Graphes
Année 2021-2022

Florent Foucaud
Malika More
Thibault Ralet
Carine Simon
Thierry Trévisan

TP2 - Arbres couvrants

Fonctions de base dans NetworkX (graphes non-orientés)

Tutoriel et documentation de NetworkX en ligne (en anglais).

```
H=nx.Graph()                      #crée un graphe
H.add_edge(0,1)                   #ajoute une arête entre les sommets 0 et 1
H.add_edges_from([(3,0),(3,4)]) #ajoute les arêtes d'une liste donnée
nx.add_path(H,[0,1,3,2])          #ajoute les arêtes du chemin 0-1-3-2
H.add_node("toto")                #ajoute un sommet nommé "toto"
H.remove_node(s)                  #supprime le sommet s
H.nodes                           #sommets du graphe (attention, pour en faire
                                   une vraie liste Python, écrire: list(H.nodes))
H.edges                           #arêtes du graphe (attention, pour en faire
                                   une vraie liste Python, écrire: list(H.edges))
H.edges(s)                        #les arêtes qui touchent le sommet s
H.order()                         #le nombre de sommets de H
H.number_of_nodes()               #idem
H.size()                          #le nombre d'arêtes de H
H.number_of_edges()               #idem
H.has_edge(s1,s2)                 #renvoie True s'il existe une arête entre s1 et s2
                                   (renvoie False sinon)
H.neighbors(s)                    #un itérateur sur les voisins du sommet s dans H
                                   pour obtenir une liste, écrire: list(H.neighbors(s))
H.degree                          #un dictionnaire dont les clés sont les sommets
                                   de H et les valeurs, leur degré
H.degree[s]                       #le degré du sommet s dans H
```

Manipulation des graphes valués/pondérés sur les arêtes

Dans ce TP on va travailler avec des graphes non-orientés mais avec des poids (entiers non-négatifs) sur les arêtes. Comme illustré ci-dessous, on peut directement créer des arêtes avec des poids, ou encore rajouter ou modifier des poids aux arêtes existantes.

```
import networkx as nx              #pour la gestion des graphes
import matplotlib.pyplot as plt    #pour l'affichage
                                   #(on les renomme pour raccourcir le code)
import random                      #pour des nombres aléatoires

G0=nx.Graph()
G0.add_edge(0,1,weight=3)          #ajoute une arête entre 0 et 1 de poids 3

RG=nx.gnp_random_graph(20,0.3)     #crée un graphe aléatoire à 20 sommets où
                                    chaque arête existe avec probabilité 0.3
                                    (la probabilité est un réel entre 0 et 1)

#on ajoute des poids aléatoires sur les arêtes (poids entre 1 et 10)
for e in RG.edges:
    RG[e[0]][e[1]]["weight"]=random.randrange(1,10)

w=RG.get_edge_data(v,w)["weight"] #obtenir le poids de l'arête entre v et w
w=RG.get_edge_data(*e)["weight"]  #obtenir le poids de l'arête e

pos=nx.spring_layout(RG)           #on définit les positions des sommets
                                   #avec l'algorithme de dessin par défaut
nx.draw(RG,pos,with_labels=True)  #on prépare le dessin du graphe
#on ajoute l'affichage des poids des sommets sur les arêtes:
nx.draw_networkx_edge_labels(RG,pos,
    edge_labels=nx.get_edge_attributes(RG,"weight"))
plt.show()                         #on affiche le dessin
```

Graphes de test pour l'affichage d'un arbre couvrant

Dans ce TP vous allez calculer des arbres couvrants, sous la forme d'un ensemble darêtes du graphe initial.

Pour tester l'algorithme, vous pouvez par exemple le faire tourner sur l'exemple de la Figure 1, pour lequel la solution est unique, et qu'on peut créer et afficher avec le code NetworkX suivant.

```
G1 = nx.Graph()
G1.add_edge(0,1,weight=5)
G1.add_edge(0,2,weight=3)
G1.add_edge(0,3,weight=3)
G1.add_edge(0,4,weight=1)
G1.add_edge(1,2,weight=2)
G1.add_edge(2,3,weight=4)
G1.add_edge(3,4,weight=3)
G1.add_edge(3,5,weight=2)
G1.add_edge(4,5,weight=1)
dico_positions = {0:(0,0),1:(2,-1),2:(1.8,-3),3:(-0.3,-2),4:(-2,-1),5:(-2.5,-3)}
nx.draw(G1,dico_positions,with_labels=True)
```

```
nx.draw_networkx_edge_labels(G1,dico_positions,
    edge_labels=nx.get_edge_attributes(G1,"weight"))
plt.show()
```

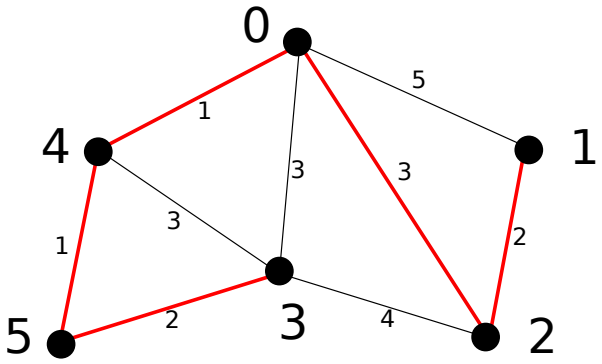

FIGURE 1 – Un graphe valué et son (unique) arbre couvrant de poids minimal en rouge.

Pour afficher un arbre couvrant, vous pouvez par exemple utiliser le code suivant qui va afficher `G` avec un ensemble d'arêtes colorées en rouge :

```
aretes_arbre = [(0,2),(1,2),(0,4),(4,5),(3,5)]
nx.draw_networkx_edges(G1,dico_positions,edgelist=aretes_arbre,
    width=5,alpha=0.5,edge_color="red")
plt.show()
```

Un autre exemple, issu du TD, est le suivant (Figure 2).

```
G2 = nx.Graph()
G2.add_edge("A", "B", weight=7)
G2.add_edge("A", "C", weight=8)
G2.add_edge("A", "J", weight=6)
G2.add_edge("B", "E", weight=4)
G2.add_edge("C", "I", weight=9)
G2.add_edge("C", "D", weight=5)
G2.add_edge("D", "E", weight=3)
G2.add_edge("D", "F", weight=5)
G2.add_edge("D", "G", weight=16)
G2.add_edge("E", "J", weight=3)
G2.add_edge("E", "K", weight=2)
G2.add_edge("F", "G", weight=8)
G2.add_edge("F", "H", weight=4)
G2.add_edge("G", "H", weight=12)
G2.add_edge("G", "J", weight=1)
G2.add_edge("G", "L", weight=12)
G2.add_edge("I", "J", weight=6)
G2.add_edge("K", "L", weight=10)
dico_positions2 = {"A":(0,0),"C":(2,0),"D":(4,0),"F":(6,0),"B":(1,-2),"E":(3,-2),
```

```
    "G":(5,-2),"H":(7,-2), "I":(1,-4), "J":(3,-4), "K":(5,-4), "L":(7,-4)}
nx.draw(G2, with_labels=True, pos=dico_positions2)
nx.draw_networkx_edge_labels(G2,dico_positions2,
    edge_labels=nx.get_edge_attributes(G2,"weight"))
plt.show()
```

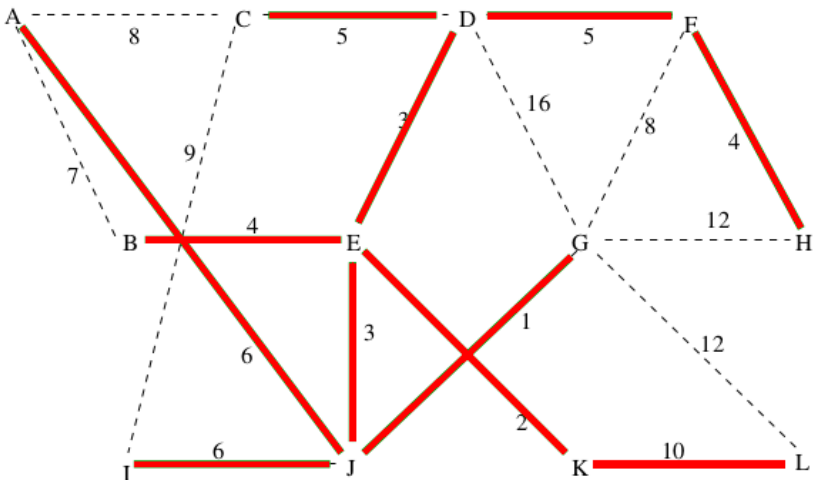

FIGURE 2 – Un deuxième graphe valué et son (unique) arbre couvrant de poids minimal en rouge.

Exercice 1 (Algorithme de Prim)**.**

L'algorithme de Prim est rappelé ci-dessous, après les questions.

1. Coder une fonction `aretes_traversantes(G,S)` qui prend en paramètre un graphe `G` et une liste `S` de sommets de `G`, et qui renvoie la liste des arêtes de `G` qui ont une extrémité dans `S` et l'autre extrémité hors de `S`.
2. Implémenter une version simplifiée de l'algorithme de Prim qui ne se soucie pas des poids des arêtes, avec une fonction `algo_prim_simple(G)` qui calcule un arbre couvrant de `G` (pas forcément de poids minimal). Pour cela, à chaque étape, l'algorithme prendra une arête quelconque parmi les arêtes autorisées et l'ajoutera à l'arbre calculé jusqu'à présent.

   La fonction renverra la liste des arêtes de l'arbre couvrant. Tester avec les graphes `G1` ou `G2` des Figures 1 et 2, puis éventuellement avec des graphes aléatoires.

   Conseil : maintenir une liste `sommets_visites` *des sommets visités et une liste des arêtes sélectionnées. À chaque étape il faudra considérer toutes les arêtes entre les sommets visités et les sommets non-visités grâce à* `aretes_traversantes(G,sommets_visites)`.
3. Implémenter l'algorithme de Prim au complet, par une fonction `algo_prim(G)` qui renvoie la liste des arêtes de l'arbre couvrant de poids minimal de `G` calculé par l'algorithme. Tester.

`Algorithme de Prim :`

- `Initialiser` T `avec` $\begin{cases} \texttt{sommets : un sommet de } G \texttt{ qu'on choisit} \\ \texttt{arêtes : aucune} \end{cases}$
- `Répéter tant que` T `ne couvre pas tous les sommets :`
  - ⋆ `Trouver toutes les arêtes de` G `qui relient un sommet de` T `et un sommet extérieur à` T
  - ⋆ `Parmi celles-ci, choisir une arête de poids le plus petit possible`
  - ⋆ `Ajouter à` T `cette arête et le sommet correspondant`
- `S'arrêter dès que tous les sommets de` G `sont dans` T
- `Retourner` T

Exercice 2 (Algorithme de Kruskal)**.**

Implémenter l'algorithme de Kruskal, rappelé ci-dessous, par une fonction `algo_kruskal(G)` qui renvoie la liste des arêtes de l'arbre couvrant de `G` calculé par l'algorithme. Tester comme à l'exercice précédent.

`Algorithme de Kruskal :`

- `Initialiser` T `avec` $\begin{cases} \texttt{sommets : tous les sommets de } G \\ \texttt{arêtes : aucune} \end{cases}$
- `Répéter tant que` T `n'est pas connexe :`
  - `Trouver toutes les arêtes de` G `qui relient deux composantes connexes de` T
  - `en trouver une de poids minimal et l'ajouter à` T
- `S'arrêter dès que` T `est connexe`
- `Retourner` T

Il y a plusieurs façons d'implémenter cet algorithme. Une difficulté est de savoir, à chaque étape, quelles arêtes on a le droit d'ajouter à la solution partielle. Pour cela, on vous propose la méthode suivante. Notez qu'à toute étape de l'algorithme, les arêtes choisies forment une forêt (un graphe sans cycle mais pas forcément connexe, c'est à dire, un ensemble d'arbres). Au début de l'algorithme on a n arbres à un sommet chacun (où n est le nombre de sommets du graphe), et à la fin, on a un seul arbre à n sommets.

On peut donc, à chaque étape, maintenir une *partition* des sommets du graphe, dont chaque partie correspond à un des arbres de la forêt. Pour cela on peut donner un numéro à chaque arbre, et attribuer le numéro de son arbre à chaque sommet du graphe. On pourra pour cela utiliser un dictionnaire `partition_sommets` dont les clés sont les sommets du graphe et les valeurs, les numéros de partie. Par exemple si on a `partition_sommets[v] == 5` cela signifie que le sommet `v` appartient à la partie numérotée `5`.

Ainsi, au tout début de l'algorithme, on a une forêt où chaque arbre a un seul sommet, et aucune arête, et donc chaque sommet a un numéro différent (on peut les numéroter de 0 à $n-1$, où n est le nombre de sommets). Il faut donc initialiser le dictionnaire de cette façon.

Au contraire, à la fin de l'algorithme, tous les sommets sont dans le même arbre couvrant, et donc ils doivent tous avoir le même numéro dans le dictionnaire.

À chaque étape, lorsqu'on considère une nouvelle arête à ajouter, il faut vérifier que ses deux extrémités ont bien deux numéros différents (sinon on crée un cycle). Si on choisit une arête entre v et w avec `partition_sommets[v] == i` et `partition_sommets[w] == j`, il faut fusionner les deux arbres, c'est-à-dire, soit donner le numéro `j` à tous les sommets numérotés `i`, ou l'inverse.

Exercice 3 (Bonus : algorithme de Borůvka)**.**
Il existe un autre algorithme d'arbre couvrant de poids minimal, l'algorithme de Borůvka. Le comprendre et l'implémenter.