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

Florent Foucaud
Malika More
Thibault Ralet
Carine Simon
Thierry Trévisan

TP5 - Flots

Fonctions de base dans NetworkX (graphes orientés)

Tutoriel et documentation de NetworkX en ligne (en anglais).
Dans ce TP quelques fonctions de bases pourraient être utiles :

```
R=nx.DiGraph()                    # crée un graphe orienté
R.add_edge("A","C",flot=[8,12]) # ajoute un arc entre les sommets A et C avec comme
                                    attribut "flot" associé la liste [8, 12]
R.nodes                           # tous les sommets du graphe (attention, pour en faire
                                    une vraie liste Python, écrire: list(H.nodes))
R.edges                           # tous les arcs du graphe (attention, pour en faire
                                    une vraie liste Python, écrire: list(R.edges))
R.degree(s)                       # degré du sommet s
R.in_degree(s)                    # degré entrant du sommet s
R.out_degree(s)                   # degré sortant du sommet s
R.neighbors(s)                    # un itérateur sur les voisins du sommet s dans R
                                    pour obtenir une liste, écrire: list(H.neighbors(s))
R.successors(s)                   # idem pour les sommets sortants
R.predecessors(s)                 # idem pour les sommets entrants
R.out_edges(s)                    # idem pour les arêtes sortantes
R.in_edges(s)                     # idem pour les arêtes entrantes
R.get_edge_data(*a)["flot"]       # valeur de l'attribut "flot" associé à l'arc a
R.edges[s1,s2]["flot"]            # accès à la valeur de l'attribut "flot" associé à l'arc
                                  # (s1,s2) tant en lecture qu'en écriture
                                  # exemple : print(R.edges["A","C"]["flot"])
                                  # ou alors : R.edges["A","C"]["flot"] = [10,12]
```

Un flot pour tester les fonctions du TP

On stocke la valeur de flot et la capacité de chaque arc dans une liste de taille 2, dans un attribut "flot".

```
G0 = nx.DiGraph()
G0.add_edge("A","C",flot=[8,12])
G0.add_edge("A","D",flot=[8,8])
G0.add_edge("A","E",flot=[3,5])
G0.add_edge("C","H",flot=[1,2])
G0.add_edge("C","F",flot=[7,7])
G0.add_edge("D","F",flot=[1,3])
G0.add_edge("D","G",flot=[7,10])
G0.add_edge("E","G",flot=[0,2])
G0.add_edge("E","J",flot=[3,3])
G0.add_edge("F","H",flot=[6,6])
G0.add_edge("F","I",flot=[2,4])
G0.add_edge("G","I",flot=[4,5])
G0.add_edge("G","J",flot=[3,5])
G0.add_edge("H","K",flot=[7,7])
G0.add_edge("I","K",flot=[2,8])
G0.add_edge("I","L",flot=[4,4])
G0.add_edge("J","L",flot=[6,6])
G0.add_edge("K","B",flot=[9,12])
G0.add_edge("L","B",flot=[10,14])

dico_positions = {"A":(0,0),"B":(5,0),"C":(1,2),"D":(1,0),
                  "E":(1,-2),"F":(2,1),"G":(2,-1), "H":(3,2),
                  "I":(3,0), "J":(3,-2),"K":(4,1),"L":(4,-1)}
nx.draw(G0,dico_positions,with_labels=True)

nx.draw_networkx_edge_labels(G0,dico_positions,
                             edge_labels=nx.get_edge_attributes(G0,"flot"))
plt.show()
```

Exercice 1 (Algorithme de Ford-Fulkerson)**.**

Pour trouver un flot maximum dans un réseau donné, on rappelle l'algorithme de Ford-Fulkerson vu en cours et TD :

— on part d'un flot quelconque (éventuellement nul) ;
— on fabrique le graphe d'écart ;
— on cherche un chemin augmentant grâce au graphe d'écart ;

et on répète ceci jusqu'à ce qu'on ne trouve plus de chemin augmentant.

Pour pouvoir implémenter cet algorithme, récupérer les primitives des files codées dans les TP précédents, et écrire les fonctions suivantes.

1. Écrire la fonction `entree(R)` qui retourne l'entrée du réseau `R` (le noeud qui n'a pas d'arc entrant).

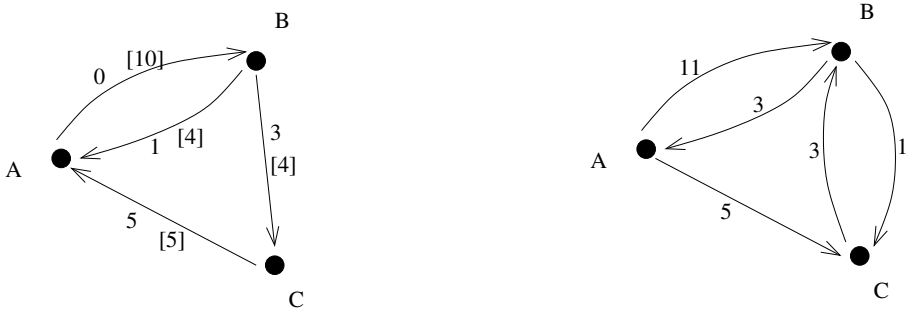
2. Écrire la fonction `sortie(R)` qui retourne la sortie du réseau `R` (le noeud qui n'a pas d'arc sortant).

3. Écrire la fonction `c(R,a)` qui retourne la capacité de l'arc `a` donné.

4. Écrire la fonction `vf(R,a)` qui retourne la valeur du flot de l'arc `a` donné.

5. Écrire la fonction `valeur_flot(R)` qui retourne la valeur du flot total pour le réseau `R`.

6. Écrire la fonction `graphe_d_ecart(R)` qui retourne un graphe orienté correspondant au graphe d'écart du réseau `R`.
   Le graphe d'écart sera un graphe orienté dans lequel est associé à chaque arc un attribut val permettant de mettre la valeur de flot possible sur cet arc. Pour rappel :
   — les sommets de ce graphe sont les mêmes que ceux de `R`
   — si dans `R` on a un arc allant du sommet `s1` au sommet `s2` de capacité `c` et de valeur de flot `f` (avec `f < c`) , alors dans le graphe d'écart il y aura un arc de `s1` à `s2` de valeur `c-f` (car on peut augmenter le flot jusqu'à atteindre la capacité)
   — si dans `R` on a un arc allant du sommet `s1` au sommet `s2` de valeur de flot `f` avec `f`$\neq 0$, alors dans le graphe d'écart il y aura un arc de `s2` à `s1` de valeur `f` (car on peut enlever du flot ce qui s'y trouve déjà)
   Attention : dans le cas où entre 2 sommets du réseau il existe deux arcs en sens inverse (exemple entre A et B sur l'exemple dessiné ci-dessous), ceux-ci peuvent donner lieu chacun à deux arcs en sens inverse dans le graphe d'écart. Ce qui donnerait entre autre deux arcs de B à A (l'un avec val à 1, et l'autre avec val à 10). Comme dans le graphe d'écart, on ne peut avoir qu'un arc de B à A sa val sera 10+1=11.
   Pour que ce genre de cas soit pris en compte, écrire la fonction `ajout_modif_dif(G, sdep, sarr, v)` que la fonction `graphe_d_ecart(R)` appellera et qui ajoute dans le graphe G l'arc allant de sdep à sarr et de val v si l'arc n'est pas déjà dans G, ou alors ajoute v à la val de l'arc si celui-ci existe déjà dans G.



Pour visualiser ce graphe d'écart (noté ici GE), on peut le faire comme pour le réseau :

```
# on utilise le même dico_position que pour le réseau
nx.draw(GE,dico_positions,with_labels=True)
nx.draw_networkx_edge_labels(GE,dico_positions,
```

```
                                    edge_labels=nx.get_edge_attributes(GE,"val"))
plt.show()
```

mais le problème, c'est que si entre 2 sommets il y a 2 arcs en sens inverse, ils seront confondus et on ne verra que l'une des deux valeurs associées s'afficher. On peut cependant faire appel à une autre librairie qui gère mieux cet affichage, et ajouter à ces lignes juste avant `plt.show()` les lignes suivantes (il faut éventuellement installer `pydot` en tapant `pip3 install pydot` dans un terminal) :

```
import matplotlib.image as mpimg
from io import BytesIO
...

GE_pydot = nx.drawing.nx_pydot.to_pydot(GE) # GE_pydot est un graph de pydot
for a in GE_pydot.get_edges(): # on ajoute les poids des arcs
    a.set_label(a.get_attributes()["val"])
png_str = GE_pydot.create_png()
sio = BytesIO() # file-like string, appropriate for imread below
sio.write(png_str)
sio.seek(0)     # on remet la tête de lecture/écriture au début
img = mpimg.imread(sio)
imgplot = plt.imshow(img)
```

7. Écrire la fonction `chemin_augmentant(E,e,s)` qui retourne un chemin augmentant de l'entrée `e` vers la sortie `s` en tenant compte du graphe d'écart `E`. Le chemin est retourné sous la forme d'une liste d'arcs (de couples de sommets).
   Cette fonction appellera la fonction `parcours_largeur_pour_chemin(E,e)` qui parcourt le graphe des écarts `E` à partir de l'entrée `e` afin de trouver pour chaque sommet par quel sommet on peut l'atteindre (son prédécesseur) et la valeur de l'arc correspondant. (Le parcours en largeur dans un graphe orienté est similaire à celui dans un graphe non orienté, sauf qu'on ne peut suivre que les arcs dans le bon sens.) Cette fonction retourne un dictionnaire qui associe à chaque sommet de `E` un couple formé du sommet qui le précède et de la valeur de l'arc.

8. Écrire la fonction `valeur_augmentation_possible(Ch)` qui retourne la valeur de l'augmentation de flot possible selon le Chemin augmentant `Ch` donné en paramètre.

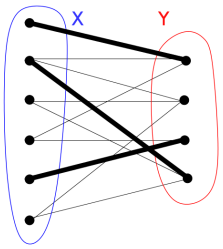
9. Écrire la fonction `augmenter_flot(R)` qui réalise une étape de l'algorithme de Ford-Fulkerson (calcule le graphe d'écart, en déduit un chemin augmentant, puis fait évoluer le flot sur le réseau selon le chemin augmentant trouvé). Cette fonction retourne un booléen indiquant si le flot a pu être augmenté.

10. Écrire la fonction `ford_fulkerson(R)` qui applique l'algorithme de Ford-Fulkerson (en cherchant à augmenter le flot tant que c'est possible) et affiche la valeur du flot trouvé.
11. (Optionnel) Modifier la fin de l'algorithme pour qu'il affiche une coupe minimale.

Exercice 2 (Couplage dans un graphe biparti)**.**

On rappelle qu'un graphe est dit **biparti** si ses sommets peuvent être partitionnés en deux ensembles X et Y tels que toute arête du graphe relie un sommet de X à un sommet de Y. (Il n'existe donc aucune arête entre 2 sommets de X, ni entre 2 sommets de Y).

Un **couplage** d'un graphe est un sous-ensemble d'arêtes du graphe deux-à-deux indépendantes, c'est-à-dire n'ayant jamais de sommet en commun (voir figure ci-dessous, le couplage sont les arêtes en gras).



Étant donné un graphe supposé biparti, on va créer un réseau de flot (voir exercice 3 du TD) en orientant toutes les arêtes de X vers Y. On rajoute un sommet d'entrée du réseau qui a un arc vers tous les sommets de X, et un sommet sortie qui a un arc depuis tous les sommets de Y. Toutes les capacités sont à 1. Un flot admissible du réseau correspond exactement à un couplage du graphe initial, et vice-versa.

1. Étant donné un graphe biparti, calculer deux listes de sommets, correspondant aux ensembles X et Y de la définition. On pourra le faire avec un parcours de graphe quelconque.
2. Écrire une fonction `couplage(G)` qui crée un réseau de flot à partir d'un graphe biparti `G` donné, avec toutes les capacités à 1, et les valeurs de flot à 0, puis trouve un flot maximum à l'aide de l'algorithme de Ford-Fulkerson, et enfin, renvoie la liste des arêtes d'un couplage maximum.
3. Tester la fonction sur des graphes bipartis, en affichant en rouge les arêtes du couplage obtenu. On peut obtenir des graphes bipartis de la façon suivante.

```
H1 = networkx.heawood_graph() #le graphe de Haawood, un graphe biparti à 14 sommets
# un graphe biparti aléatoire avec X et Y de taille n et m, où chaque arête
# existe avec probabilité p :
H2 = networkx.bipartite_random_graph(n, m, p, seed=None, directed=False)
```