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

Florent Foucaud
Malika More
Thibault Ralet
Carine Simon
Thierry Trévisan

TP3 et TP4 - Parcours de graphes et distances

Fonctions de base dans NetworkX (graphes non-orientés)

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

```
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
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.neighbors(s)                    #un itérateur sur les voisins du sommet s dans H
                                   pour obtenir une liste, écrire: list(H.neighbors(s))
H.nodes[s]["attri"]               #accède à l'attribut nommé "attri" du sommet s
                                   (tant en lecture qu'en écriture)
                                   exemple: H.nodes[s]["attri"]=2
                                   ou alors print(H.nodes[s]["attri"])
```

Exercice 1 (La frontière)**.**

Pour parcourir un graphe, l'important est la structure de données qui stocke la frontière, c'est-à-dire l'ensemble des sommets visités mais pas encore entièrement explorés. Dans cet exercice, vous allez coder des fonctions permettant de réaliser les opérations élémentaires sur cette structure de données : ajouter un élément, enlever un élément, tester si la structure de données est vide, accéder au prochain élément dans la structure de données.

Dans tous les cas, on vous demande d'utiliser une liste `Python3`**.** Utilisez les méthodes `append` et `pop` pour ajouter et enlever des éléments d'une liste. On écrit par exemple `L.append(x)` pour ajouter un élément `x` en fin de liste `L`. Par ailleurs, `L.pop(i)` retire et

renvoie l'élément à l'indice `i` de la liste `L` (`L.pop(-1)` retire le dernier élément). On peut aussi utiliser `del L[i]` pour supprimer l'élément de `L` à l'indice `i`. Le nombre d'éléments d'une liste `L` est donnée par `len(L)`.

1. Quelle est la structure de données qui permet d'effectuer un parcours en profondeur ? Un parcours en largeur ?
2. Pour une file `F`, et un élément `v`, la fonction ci-dessous ajoute l'élément `v` à la file `F`.

   ```
   def ajouter_file(F,v):
     F.append(v)
     return
   ```

   Écrire les fonctions `enlever_tete_file(F)`, `est_file_vide(F)`, `valeur_tete_file(F)`.
3. Pour une pile `P`, et un élément `v`, la fonction ci-dessous ajoute l'élément `v` à la pile `P`.

   ```
   def ajouter_pile(P,v):
     P.append(v)
     return
   ```

   Écrire les fonctions `enlever_sommet_pile(P)`, `est_pile_vide(P)`, `valeur_sommet_pile(P)`.

Exercice 2 (Les parcours en largeur et en profondeur)**.**

On rappelle l'algorithme de parcours générique, vu en cours.

Parcours du graphe G à partir du sommet ''source'' s
- V est la liste des sommets visités. Contient initialement seulement s.
- F est la frontière : les sommets visités mais qui ont peut-être encore des voisins non visités. Contient initialement seulement s.
- Répéter tant que la frontière F est non-vide :
  - ⋆ Considérer un sommet x de la frontière F.
  - ⋆ Si il existe un voisin y de x pas encore dans V :
    - ajouter y dans V et dans F
  - ⋆ Sinon :
    - enlever x de F
- Retourner V

- Dans le parcours en largeur, la frontière F est une file. Pour considérer un sommet de la frontière on regarde toujours la tête de file. Pour enlever un sommet on défile, et pour en ajouter un, on enfile. Ainsi, l'algorithme inspecte tous les voisins d'un sommet avant de passer au suivant.
- Dans le parcours en profondeur, la frontière F est une pile. Pour considérer un sommet de la frontière on regarde toujours le haut de la pile. Pour enlever un sommet on dépile, et pour en ajouter un, on empile. Ainsi, l'algorithme va de sommet en sommet jusqu'à être bloqué et revenir sur ses pas.

À l'aide des fonctions écrites dans le premier exercice :

1. écrire la fonction `parcours_largeur(G,s)` qui effectue un parcours en largeur du graphe `G` à partir du sommet `s`, et qui retourne la liste des sommets visités.

2. écrire la fonction `parcours_profondeur(G,s)` qui effectue un parcours en profondeur du graphe `G` à partir du sommet `s`, et qui retourne la liste des sommets visités.

Ne pas oublier de tester vos programmes, par exemple sur le graphe `G` suivant :

```
####################
G=nx.Graph()
G.add_edges_from([(1,2),(1,5),(2,3),(2,5),(3,4),(4,5),(4,6),(5,0)])

nx.draw(G,with_labels=True)
plt.show()
####################
```

Exercice 3 (Variante : l'incendie)**.**

On veut simuler la propagation d'un incendie dans un bâtiment. On représente le bâtiment par un graphe, dont les sommets sont les salles, et une arête entre deux salles indique qu'elles communiquent, et que le feu va se propager de l'une à l'autre. On considère que chaque minute, le feu envahit toutes les salles qui communiquent avec une salle en feu ; les salles qui sont en feu continuent de brûler indéfiniment.

1. Écrire une fonction `propagation(G,E)` qui, à partir d'un graphe `G` et d'un ensemble `E` de sommets représentant des salles en feu, donne l'ensemble `F` des salles qui seront en feu 1 minute plus tard.
2. Écrire une fonction `propagation_temps(G,E,t)` qui indique quelles salles seront en feu dans `t` minutes si `E` est l'ensemble des salles en feu maintenant.
3. Si une seule salle est en feu dans le bâtiment, quel est l'ensemble des salles qui prendront feu à un moment ou à un autre ?
4. Représenter graphiquement la propagation du feu à partir d'une salle. Par exemple, on peut afficher en rouge les sommets en feu en utilisant la fonction `nx.draw_networkx_nodes(G, pos, nodelist=L, node_color="red")` où `pos` est le dictionnaire de positions et `L` est la liste des sommets à afficher en rouge.

Exercice 4 (Variante : les trésors)**.**

Le code ci-dessous a pour effet de créer un graphe `G1` avec des trésors sur les sommets (un trésor est un entier).

```
#########################
G1=nx.Graph()
G1.add_edges_from([("a","f"),("a","b"),("f","c"),("f","e"),("f","d"),
                   ("c","e"),("c","b"),("c","d"),("e","d")])
G1.nodes["a"]["tresor"]=3
G1.nodes["f"]["tresor"]=4
G1.nodes["c"]["tresor"]=5
G1.nodes["e"]["tresor"]=1
```

```
G1.nodes["d"]["tresor"]=2
G1.nodes["b"]["tresor"]=0

## pour afficher le graphe avec les trésors associés à chaque sommet
dico_positions_sommets = {"a":(0,0),"b":(0,2),"c":(2,2),
                          "d":(4,2),"e":(4,0),"f":(2,0)}
dico_positions_tresors = {"a":(0,-0.1),"b":(0,2.1),"c":(2,2.1),
                          "d":(4,2.1),"e":(4,-0.1),"f":(2,-0.1)}
nx.draw(G1,dico_positions_sommets,with_labels=True)
nx.draw_networkx_labels(G1,dico_positions_tresors,
                        labels=nx.get_node_attributes(G1,"tresor"))
plt.show()
#########################
```

1. Écrire une fonction `tresor(G,s)` prenant en entrée un graphe `G` et un sommet `s` et renvoyant le trésor présent sur le sommet `s`.
2. Écrire une fonction `parcours_tresor(G,s)` qui prend en entrée un graphe `G` et un sommet `s` de départ, et parcourt le graphe en choisissant de traiter à chaque étape le sommet de la frontière (déjà connu mais non traité) ayant le plus grand trésor. La fonction doit renvoyer la liste des sommets traités dans l'ordre de traitement.

Exercice 5 (Distances à un sommet)**.**

1. En adaptant l'algorithme de parcours en largeur, écrire une fonction `distances(G,s)` qui, pour un graphe `G` connexe et un sommet `s` donnés, calcule la distance (en nombre d'arêtes) entre `s` et chaque sommet de `G`. Cette fonction retourne un dictionnaire associant une distance à chaque sommet.
2. (optionnel) Écrire une fonction `affiche_plus_courts_chemins_a_partir_de(G,s)` qui, pour un graphe `G` connexe et un sommet `s` donnés, affiche pour chaque sommet du graphe un plus court chemin à partir de `s`.
   Cette fonction appellera la fonction `plus_courts_chemins(G,s)` qui calcule pour chaque sommet son prédécesseur (le sommet par lequel il est découvert lors du parcours à partir de s).
   Exemple : `affiche_plus_courts_chemins_a_partir_de(G,1)` (avec G de l'exercice 2) donnera :

```
1
2 <- 1
5 <- 1
3 <- 2 <- 1
4 <- 5 <- 1
6 <- 4 <- 5 <- 1
0 <- 5 <- 1
```

Exercice 6 (Connexité)**.**

1. Écrire une fonction `est_connexe(G)` qui détermine si un graphe `G` est connexe. Utiliser pour cela un algorithme parcours codé dans l'exercice 2.
2. Écrire une fonction `composantes_connexes(G)` qui calcule et retourne la liste des composantes connexes d'un graphe `G`.

Tester vos fonctions, par exemple avec le graphe `G` précédent et le graphe `H` suivant :

```
############################
H=nx.Graph()
H.add_nodes_from([1,2,3,4,5,6,7,8,9,10,11])
H.add_edges_from([(1,2),(1,5),(2,3),(2,5),(4,6),(5,0),(8,11),(9,8),(10,8)])

nx.draw(H,with_labels=True)
plt.show()
############################
```

Exercice 7 (Parcours d'arbres binaires)**.**

Un *arbre binaire* est un arbre qui possède une unique *racine* (un sommet particulier) et où chaque sommet a au plus deux sommets voisins spéciaux, appelés ses *fils*. S'il a deux fils, l'un est le *fils gauche* et l'autre est le *fils droit*. Un sommet est appelé le *père* de son fils. Ainsi, chaque sommet de l'arbre binaire a au maximum trois voisins (deux fils et un père). La racine n'a pas de père, et les *feuilles* sont les sommets sans fils.

Dans cet exercice, on s'intéresse au parcours en profondeur d'un arbre binaire `T` à partir de sa racine `r`. Un certain traitement doit être effectué une fois et une seule sur chaque sommet au cours de ce parcours. Ici, le traitement consiste simplement à afficher le nom du sommet.

Vous pourrez tester vos programmes sur l'arbre `ArbreBin` ci-dessous, dont la racine est le sommet `1`.

```
######################
ArbreBin=nx.Graph()
ArbreBin.add_edges_from([(1,2),(1,5),(3,4),(4,5),(4,6),(5,0)])

# dessiné comme ceci
nx.draw(ArbreBin,with_labels=True)
plt.show()

# ou comme cela
dico_pos_arbre = {1:(0,0),2:(-1,-1),5:(2,-1),0:(1,-2),4:(3,-2),3:(2,-3), 6:(4,-3)}
nx.draw(ArbreBin,dico_pos_arbre,with_labels=True)
plt.show()
######################
```

1. Dessiner sur une feuille un arbre binaire comportant une douzaine de sommets. Effectuer à la main un parcours en profondeur de cet arbre, en choisissant par convention d'abord le fils gauche d'un sommet si ce dernier possède deux fils.

2. Pendant le parcours en profondeur d'un arbre binaire, combien de fois et à quelles occasions chaque sommet intérieur (c'est-à-dire qui n'est pas une feuille) est-il visité ? Que se passe-t-il pour les feuilles de l'arbre ?
3. Écrire une fonction qui affiche le nom de chaque sommet au moment où il est ajouté à la pile. Ce type de parcours en profondeur d'un arbre binaire s'appelle un parcours **préfixe**.
4. Écrire une fonction qui affiche le nom de chaque sommet au moment où il est supprimé de la pile. Ce type de parcours en profondeur d'un arbre binaire s'appelle un parcours **postfixe**.
5. Écrire une fonction qui affiche le nom de chaque sommet la première fois qu'on le rencontre s'il n'a qu'un fils et la deuxième fois s'il en a deux. Ce type de parcours en profondeur d'un arbre binaire s'appelle un parcours **infixe**.
6. L'un de ces trois parcours (préfixe, postfixe, infixe) correspond à la notation polonaise inversée vue en TD. Lequel ?

Exercice 8 (Algorithme de Dijkstra - distances dans les graphes valués)**.**

Dans cet exercice, on considère des graphes valués, c'est-à-dire que les arêtes possèdent un **poids** positif ou nul. La *distance* entre deux sommets est la plus petite somme des poids des arêtes parmi les chaînes qui relient les deux sommets. Ces poids sont représentés par l'attribut `weight` des arêtes du graphe dans la librairie `networkx` de Python. De plus, on crée un attribut `distance` pour les sommets qui représentera la distance à la source. Les distances sont d'abord toutes initialisées à `None` comme dans l'exemple ci-dessous :

```
##################################
##Un graphe pour tester Dijkstra
##################################
print("Un graphe pour tester Dijkstra")
G=nx.Graph()
G.add_nodes_from(["A","B","C","D","E","F","G","H","I","J"],distance=None)
print(G.nodes())
G.add_edges_from([("A", "B", {"weight": 4}),("A", "C", {"weight": 2}),
                  ("A", "E", {"weight": 1}),("B", "F", {"weight": 3}),
                  ("C", "G", {"weight": 1}),("C", "H", {"weight": 2}),
                  ("D", "H", {"weight": 1}),("E", "J", {"weight": 5}),
                  ("F", "I", {"weight": 2}),("I", "J", {"weight": 5}),
                  ("H", "J", {"weight": 6})])
print(G.edges())
#########################

print(G.edges[("A","B")]["weight"])
print(G.nodes["A"]["distance"])
########################
```

On rappelle ci-dessous l'algorithme de Dijkstra qui calcule les distances à un sommet source donné :

Algorithme de Dijkstra pour le graphe G à partir du sommet source s

- L représentera la frontière. Contient initialement seulement s.
- La valeur de distance d est initialisée à d(s)=0 et à d(v)=∞ pour chaque autre sommet v
- Une liste T contient les sommets qui ont été complètement traités. Initialement T est vide.
- Tant que L n'est pas vide :
  - ⋆ choisir un sommet v dans L qui a une valeur de distance d(v) minimale
  - ⋆ pour tout voisin w de v qui n'est pas dans T :
    - si d(v) plus le poids p de l'arête (v,w) est inférieur à d(w), on fixe d(w)=d(v)+p
    - ajouter w à L
  - ⋆ enlever v de L, ajouter v à T

Implémentez l'algorithme en suivant les étapes suivantes :

1. Écrire une fonction `mise_a_jour_voisin(G,n,v)` qui met à jour la distance du sommet `v` à la source (`v` est supposé voisin de `n` dans le graphe `G`) à partir de celle du sommet `n` (en supposant que la distance du sommet `n` à la source est un nombre).
2. Écrire une fonction `choix_prochain_sommet(G,L)`, qui renvoie le sommet de la frontière `L` qui sera choisi pour la prochaine itération de l'algorithme de Dijkstra.
3. Écrire une fonction `Dijkstra(G,s)` qui détermine les distances entre le sommet `s` et tous les autres sommets dans le graphe valué `G`.

Exercice 9 (Algorithme de Floyd-Warshall - calcul de toutes les distances)**.**

Coder et tester l'algorithme de Floyd-Warshall, rappelé ci-dessous :

Algorithme de Floyd-Warshall pour le graphe G

- Les sommets sont ordonnés : s_1 ... s_n avec n le nombre de sommets
- On initialise une matrice D de taille $n \times n$, où $D(i,j)$ devra contenir la distance entre le sommet s_i et le sommet s_j. Pour toute arête entre s_i et s_j de poids p_{ij}, on fixe $D(i,j) = p_{ij}$, et pour tout i on fixe $D(i,i) = 0$. Dans les autres cas, on fixe $D(i,j) = \infty$.
- Pour k allant de 1 à n :
  - Pour i allant de 1 à n :
    - Pour j allant de 1 à n :
      $D(i,j) = \min\{D(i,j), D(i,k) + D(k,j)\}$
- Renvoyer D

Pourquoi n'est-t-il pas nécessaire de copier la matrice D à chaque étape ? Vérifier que l'algorithme ne va pas écraser des données importantes en cours de route.