Aide-mémoire NetworkX

Network est une librairie très riche, on peut faire beaucoup de choses avec et de plusieurs manières différentes. Dans ce document, on propose quelques méthodes qui peuvent servir dans le TP. Vous êtes libres de les suivre ou de faire autrement, tant que cela utilise des fonctions de NetworkX.

Lien vers la documentation de la librairie

Selon les cas, les fonctions peuvent renvoyer des structures de données variées : des graphes bien sûr, mais aussi des listes, des dictionnaires, … Si vous avez un doute, exécutez la fonction pour voir la "tête" du résultat. Quand le résultat est un générateur (`<...generator...>`) ou un itérateur (`<...iterator...>`) Python, utilisables dans une boucle for, vous pouvez le transformer en liste (`list(générateur_ou_itérateur)`) pour voir à quoi les éléments individuels ressemblent.

Certaines fonctions s'appliquent aux objets graphes, d'autres sont des fonctions du module `nx` et reçoivent un graphe en paramètre.

Conventions dans la suite :

- Le module `networkx` est importé avec l'alias `nx`
- Les variables contenant des IDs de noeuds ont pour nom n, n1, n2
- Les variables contenant des objets graphes ont pour nom g, g1, g2
- Les propriétés (au sens *property graph*) des noeuds ou d'arcs sont appelés attributs

Importer la librairie

```
import networkx as nx
```

Créer un graphe vide

```
g = nx.Graph()      # Graphe non dirigé
g = nx.DiGraph()    # Graphe dirigé
```

Il y a d'autres sortes de graphes, ce sont les 2 principales.

Ajouter des noeuds

```
g.add_node(n, attr1=v1, attr2=v2, …)
g.add_nodes_from(ensemble)
```

Avec la première forme, on peut ajouter des attributs à la volée avec des paramètres nommés (`attr1`, `attr2` dans l'exemple).

Avec la deuxième forme, `ensemble` peut être une liste d'ID de noeuds, ou de couples avec des attributs, ou encore un dictionnaire…

Ajouter des arcs

```
g.add_edge((n1, n2), attr1=v1, attr2=v2, …)
g.add_edges_from(ensemble)
```

Avec la première forme, on peut ajouter des attributs à la volée avec des paramètres nommés (`attr1`, `attr2` dans l'exemple).

Avec la deuxième forme, `ensemble` peut être une liste de couples (tuples) d'IDs de noeuds, ou de triplets avec des attributs, ou encore un dictionnaire…

Accéder aux noeuds

```
g.nodes
```

Renvoie une collection de noeuds.

```
g.node[n]
```

Renvoie le noeud d'ID n ; c'est un dictionnaire contenant les attributs de n. On peut lire ou écrire dans ce dictionnaire.

Accéder aux arcs

```
g.edges
```

Renvoie une collection d'arcs.

```
g.edges[(n1, n2)]
```

Renvoie l'arc entre n1 et n2 s'il existe ; c'est un dictionnaire contenant les attributs de l'arc. On peut lire ou écrire dans ce dictionnaire.

Ecrire des attributs de noeuds

Individuellement :

```
g.nodes[n]['attr'] = value
g.edges[(n1, n2)]['attr'] = value
```

En masse :

```
nx.set_node_attributes(g, attrs)
nx.set_edges_attributes(g, attrs)
```

Dans ces fonctions, `attrs` est un dictionnaire dont les clefs sont les IDs de noeuds (`n`) ou d'arcs (`(n1, n2)`) à modifier, et les valeurs sont elles-mêmes des dictionnaires avec les valeurs d'attributs à écrire. Les nouveaux attributs modifient ou ajoutent d'éventuels attributs existants.

Degrés des noeuds

Pour un noeud donné :

```
g.in_degree(n)
g.out_degree(n)
g.degree(n)
```

Les versions `in_` et `out_` s'appliquent aux graphes dirigés uniquement. Ce sont les degrés entrant et sortant, respectivement. La version `g.degree()` s'applique à tout type de graphe (pour un graphe dirigé, c'est la somme des degrés entrant et sortant).

Pour tout le graphe :

```
g.in_degree()
g.out_degree()
g.degree()
```

Quand on ne précise pas de noeud, on obtient les degrés de tous les noeuds du graphe.

Parcours et chemins

```
nx.all_simple_paths(g, n1, n2)
```

Renvoie un itérateur sur tous les chemins sans cycle reliant n1 à n2.

```
nx.dfs_tree(g, n)
nx.bfs_tree(g, n)
```

Renvoie un sous-graphe de g, qui est un arbre construit en le parcourant à partir de n, avec la stratégie DFS ou BFS selon la variante.

Fusion de graphes

```
nx.compose(g1, g2)
```

L'ensemble des noeuds / arcs du résultat est l'union des noeuds / arcs de g1 et g2, et les attributs des éléments en commun sont fusionnés.

Récupération des composantes connextes

```
nx.strongly_connected_components(g)
nx.weakly_connected_components(g)
```

Il y a 2 définitions possibles pour les composantes connexes, d'où les 2 fonctions :

- La version forte ("*strongly*") met dans une même composante les noeuds qui sont entièrement connectés entre eux (il y a des arcs entre chaque paire de noeuds de la composante)
- La version faible ("*weakly*") met dans une même composante les noeuds qui sont reliés entre eux par au moins un chemin

Centralité "betweenness"

```
nx.betweenness_centrality(g, normalized, endpoints, weight, …)
```

Renvoie la centralité de tous les noeuds. Les paramètres autres que g sont facultatifs. Il y en a d'autres ; voici la signification de ceux mentionnés ci-dessus :

- `normalized` est un booléen (défaut : `False`) qui dit s'il faut diviser toutes les centralités par une constante de normalisation (cela ne change pas le classement)
- `endoints` est un booléen (défaut : `False`) qui dit s'il faut inclure les points de départ et d'arrivée des plus courts chemins lorsqu'on les compte pour évaluer la centralité
- `weight` est une chaîne de caractère avec le nom d'un attribut d'arcs pour les pondérer (interprété comme une distance dans l'évaluation des chemins). Par défaut, pas de pondération

Visualisation

```
nx.draw(g, pos, node_size, node_color, alpha, with_labels, …)
```

Dessine le graphe avec Matplotlib. Là encore seul g est obligatoire et voici les autres paramètres intéressants :

- `pos` : position 2D des noeuds (de préférence calculée par une fonction de *layout*). Par défaut, un layout "*spring*" est utilisé
- `node_size` : taille des noeuds (représentés sous forme de cercles). La valeur par défaut est élevée. Cela peut être un dictionnaire pour la faire varier par noeud
- `node_color` : nom de couleur des noeuds (bleu par défaut). Cela peut aussi être un dictionnaire
- `alpha` : transparence (un flottant entre 0 et 1) pour les noeuds et les arcs. Par défaut, pas de transparence : s'il y a beaucoup de recouvrement, on ne distingue pas très bien…

- `with_labels` : affiche les IDs de noeuds à côté des cercles. Vaut `False` par défaut

```
nx.spring_layout(g, k)
```

Calcule les positions de noeuds avec un algorithme qui considère les arcs comme des “ressorts”, de sorte que les noeuds se répartissent sur le plan. Cela améliore la lisibilité pour certains types de graphe, en limitant les recouvrements de noeuds et les croisements d’arcs. La force des ressorts est contrôlée par le paramètre `k`, un flottant.

C’est le *layout* par défaut de `nx.draw()`, mais en l’utilisant explicitement, on a plus de contrôle via les paramètres de la fonction. De plus, quand on dessine plusieurs fois un même graphe, il est intéressant de calculer les positions une fois pour toutes : cela évite des calculs parfois longs, et les noeuds ne changent pas de place d’une image à l’autre.