Informatique — MPSI/PCSI, 2023–2024

TP nº 12 – Graphes - Partie 2/3

Objectifs :

- Implémenter des graphes à l'aide d'un dictionnaire.
- Manipuler les graphes avec ces représentations.
- Implémenter les parcours en largeur, en profondeur, best-first
- Implémenter des applications de ces parcours (connexité, composantes connexes, distances...)

Rappel :

- Enregistrez dès le début du TP votre travail avec un nom de la forme `VOTRENOM_TPX.py` (évitez les accents et caractères spéciaux).

CONSIGNE IMPORTANTE :

- Pour ce TP, l'IDE SPYDER sera IMPÉRATIVEMENT utilisé.

1 Visualisation et parcours en largeur

1.1 Visualisation (partie facultative)

Le fichier `cadeau.py` fournit :

1. Le graphe test `G1`, graphe support du cours pour les parcours en largeur et en profondeur.
2. Un second graphe test `G2`.
3. Une procédure `trace_graphe_adj`, prenant en argument le dictionnaire associé au graphe, qui permet de visualiser le graphe.
4. Une procédure `trace_graphe_adj_cc`, prenant en argument le dictionnaire associé au graphe, la couleur de chaque nœud étant la valeur associée à la clé `'coul'` du dictionnaire associé à l'étiquette de chaque sommet.

Exercice 12.1
Tester les deux procédures sur les graphes tests G1 et G2 fournis.

1.2 Visualisation du parcours en largeur (ou BFS, pour Breadth First Search)

Exercice 12.2

1. Reprendre la fonction `exploration_composante` du TP n°11, également fourni dans le fichier `cadeau.py`, et la modifier afin de visualiser les différentes étapes du parcours de la composante connexe à partir du sommet de départ `r`.
   Le code couleur utilisé sera le suivant : JAUNE ("yellow") : couleur des sommets non explorés, BLEU ("blue") : couleur des sommets explorés (mais pas tous ses voisins), ROUGE ("red") : couleur des sommets explorés avec tous les voisins explorés.
   Remarque : afin de visualiser le parcours étape par étape, il est possible d'insérer la commande `input("Presser sur une touche pour poursuivre le parcours")` qui permet de stopper l'exécution du parcours tant que l'utilisateur n'a pas appuyé sur une touche. Dans le cas contraire, les différentes étapes sont affichées successivement par `Spyder` à la vitesse d'exécution de l'algorithme. Néanmoins les différents graphiques sont enregistrés et il est possible de les examiner un par un après exécution.
2. Utiliser votre fonction pour explorer toute la composante connexe du sommet `A` du graphe `G1`. Comparer vos résultats à ceux obtenus lors du parcours manuel en cours.
3. Utiliser votre fonction pour explorer toute la composante connexe du sommet `A` du graphe `G2`.

4. A partir de la fonction `exploration_composante`, écrire une nouvelle fonction `bfs2` qui prend comme arguments un dictionnaire `G` associé à un graphe et deux sommets `sd` et `sa` et qui implémente un parcours en largeur de `G` en partant du sommet de départ `sd` jusqu'au sommet d'arrivée `sa`.
   Remarque 1 : le nœud d'arrivée `sa` doit appartenir à la composante connexe du nœud de départ `sd`.
   Remarque 2 : Pour l'exploration d'une composante connexe d'un graphe `G`, depuis un sommet de départ `sd` jusqu'au sommet d'arrivée `sa` , on utilisera un dictionnaire de parcours de la forme :

```
P = {
  <sommet>: {
    "adj": [<liste des sommets adjacents>],
    "vu": <étiquette, 1 si vu>,
    "d": <distance, entre nombre de noeuds depuis la racine>,
    "p": <sommet parent lors du parcours>,
    },
  <autre sommet>: ...
}
```

   qui permet de garder une mémoire du parcours, et que l'on aura initialisé comme une copie du graphe `G` :

```
from copy import deepcopy

P = {s:deepcopy(G[s]) for s in G}
```

5. Définir une fonction `chemin` qui prend comme arguments un dictionnaire de parcours `P` associé à un graphe et deux sommets `sd` et `sa` et qui renvoie le chemin sous forme de liste de sommets de `sd` à `sa`.
6. Tester votre fonction sur le parcours en largeur `P1` du graphe `G1` du nœud `"A"` au noeud `"F"` :

```
>>> chemin(P1,"A","F")
['A', 'B', 'D', 'F']
```

   puis sur le parcours `P2` du graphe `G2` du nœud `"A"` au noeud `"L"` :

```
>>> chemin(P2,"A","L")
['A', 'B', 'C', 'J', 'L']
```

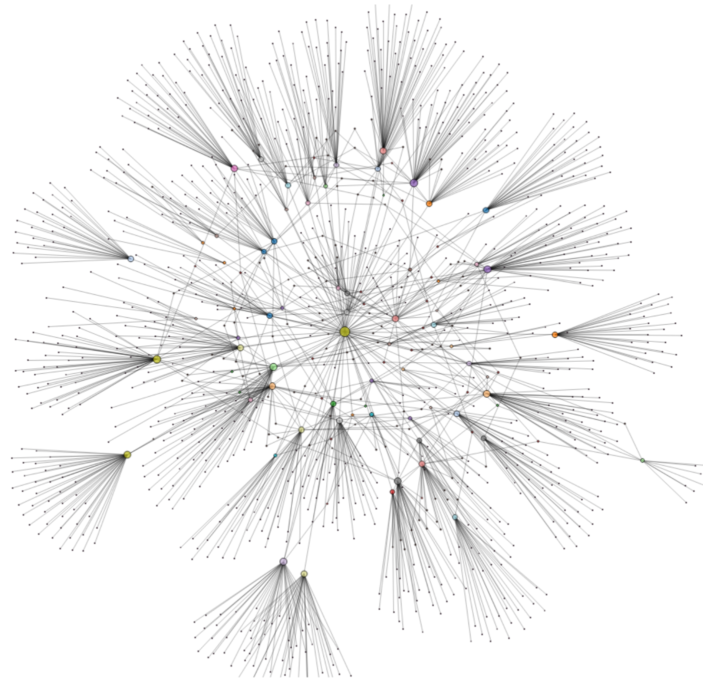
7. Écrire une fonction `estconnexe` qui prend en argument un graphe quelconque `G` et qui, en se basant sur un parcours en largeur, renvoie `True` si le graphe est connexe et `False` sinon. La tester sur les graphes tests `G1` et `G2`.
8. Écrire une fonction `cc` qui prend en argument un graphe quelconque `G` et qui renvoie les composantes connexes du graphe. A cette occasion, ***une version récursive de l'exploration d'une composante pourra éventuellement être implémentée***. La tester sur le graphe test `G1`.
9. Tester de nouveau la fonction `cc` sur le graphe test `G1` après avoir rajouté un nouveau sommet isolé `"I"` à l'aide de la fonction `sommet` du TP n°11.

1.3 Parcours en profondeur (ou DFS, pour Depth First Search)

Exercice 12.3

1. En utilisant les fonctions de visualisation du fichier `cadeau.py`, écrire une fonction `dfs`, qui prend en argument un graphe **quelconque** `G` et l'étiquette d'un nœud de départ `sd`, qui implémente un parcours en profondeur **itératif** de `G` en partant du sommet de départ `sd`.
   Le code couleur utilisé sera le suivant : JAUNE ("yellow") : couleur des sommets non explorés, BLEU ("blue") : couleur des sommets explorés.
2. Modifier la fonction `dfs`, qui prend en argument un graphe **quelconque** `G` et l'étiquette d'un nœud de départ `sd`, qui implémente un parcours en profondeur **itératif** de `G` en partant du sommet de départ `sd`, et qui renvoie la liste des sommets parcourus pour chaque composante connexe explorée.
3. Tester ces fonctions sur les graphes tests `G1` et `G2`.
4. Pour le graphe `G1`, dans quel ordre sont traités les sommets en partant de `"A"` ? Pour le graphe `G2`, dans quel ordre sont traités les sommets en partant de `"D"` ?

2 Application : un graphe de collaboration



On étudie un graphe de collaboration mathématique[1]. Il contient 6927 sommets correspondant à des chercheurs. Deux chercheurs sont reliés lorsqu'ils ont écrit un article ensemble (collaboré). Copier les fichiers `Erdos02` et `Erdos02_nodename.txt` dans votre répertoire de travail. Recopier le code suivant, qui permet de récupérer le dictionnaire d'adjacence du graphe.

```
from pickle import load
f = open("Erdos03","rb")
dErd = load(f)
f.close()
```

Le fichier `Erdos03_nodename.txt` contient les noms des auteurs : le nom en ligne i est le nom de l'auteur d'indice $i-1$ dans le graphe.

Exercice 12.4

1. Afficher le dictionnaire d'adjacence `"dErd"`. Ecrire une fonction `conversion` qui prend en argument un dictionnaire d'adjacence `"dErd"` et qui renvoie le dictionnaire complet `"G"` associé au graphe.
2. Recopier et commenter les fonctions suivantes. Que font-elles ? Quel est le nom du chercheur de sommet numéro 42 ? Quel est le sommet correspondant à ERDOS, PAUL ?

```
def numtonom(i):
  f = open("Erdos03_nodename.txt")
  lines = f.read().split('\n')
  f.close()
  return lines[i]
```

```
def nomtonum(nom):
  f = open("Erdos03_nodename.txt")
  lines = f.read().split('\n')
  f.close()
  for i in range(len(lines)):
    if lines[i]==nom: return i
  return False
```

3. Qui a le plus de collaborations ? Qui est le second auteur avec le plus de collaborations ?
   Remarque : plusieurs mathématicien(ne)s peuvent avoir le même nombre de collaboration(s).
   Questions subsidiaire, uniquement s'il reste du temps à la fin du TP : généraliser et écrire une fonction `max_collab(G,n)` qui prend en argument le graphe de collaboration, le nombre de niveaux souhaités et qui renvoie, pour chaque niveau, une liste de liste, le premier élément de la liste étant les mathématicien(ne)s concerné(e)s, le second, le nombre de collaborations.

1. d'après un exercice de TP proposé par François Giraud et Laurent Dietrich

4. Afficher la liste des noms des chercheurs sans collaborateurs.
5. Si l'on retire Paul Erdős de sa composante connexe, combien de composantes connexes obtient-on ? (on dit que Paul Erdős est un *point d'articulation).*
6. On définit la *distance de collaboration* entre deux mathématiciens comme le nombre d'arêtes du plus court chemin les reliant dans ce graphe (on la pose comme infinie s'il n'existe pas de tel chemin). Le parcours en largeur est un bon outil pour mesurer ces distances car on peut faire l'observation suivante : à l'origine, seule la racine est dans la file, à distance 0 d'elle-même. On la traite et on enfile tous ses voisins : les sommets à distance 1. On enfilera alors, en les visitant à partir d'eux, que des sommets à distance 2 de la racine etc. Ainsi, au sein d'un parcours en profondeur, la distance à la racine d'un sommet nouvellement découvert vaut toujours 1 de plus que la distance de son père.

   En se basant sur un parcours en largeur, écrire une fonction `ec_dist` prenant en argument un graphe `G` et un sommet `sd`, et qui renvoie un dictionnaire de distances, où chaque sommet de la composante connexe est une clé et sa distance au sommet de départ `sd` sa valeur. La tester sur `Erd03` en partant de ERDOS, PAUL. Que constatez-vous ?

3 Parcours best-first (s'il reste du temps)

L'algorithme **best-first** n'utilise pas une file d'attente comme l'algorithme de parcours en largeur. Parmi les sommets explorés (couleur bleue ou étiquette `"vu"` égale à 1), il choisit non pas le plus anciennement introduit dans la file, mais celui qui est le plus proche, à "vol d'oiseau" (i.e. le plus proche dans la représentation planaire du graphe), du sommet d'arrivée. On parle d'algorithme informé car il utilise une heuristique simple, reposant sur le calcul de distance pour effectuer ses choix. Pour ce faire, on a écrit une fonction `extrait_plus_proche` qui réalise une recherche de minimum dans la liste des sommets explorés. Une fois trouvé ce minimum, il attribue la valeur `2` à la clé `"vu"` du dictionnaire associé au sommet exploré correspondant.

Exercice 12.5

1. Implémenter la fonction `extrait_plus_proche`, qui prend en argument un graphe `G`, l'étiquette du sommet d'arrivée `sa` et qui renvoie l'étiquette du sommet exploré le plus proche de la "cible" (le sommet d'arrivée`sa`).
   Remarque : les distances euclidiennes entre un sommet quelconque du graphe et la cible seront calculées à partir des coordonnées des sommets. Ces coordonnées sont implémentées via une liste, valeur de la clé `"XY"` du dictionnaire (valeur) associé à chaque sommet (clé) du graphe `"XY"`.
2. Écrire une fonction `bf` qui prend comme arguments un dictionnaire `G` associé à un graphe et deux sommets `sd` et `sa` et qui implémente le parcours **best-first** de `G` en partant du sommet de départ `sd` jusqu'au sommet d'arrivée `sa`.
   Remarque : le nœud d'arrivée `sa` doit appartenir à la composante connexe du nœud de départ `sd`.
3. Tester ces fonctions sur le graphe test `G2`, en partant du sommet `"H"` jusqu'au sommet `"F"`.
4. Tester ces fonctions sur le graphe test `G2`, en partant du sommet `"A"` jusqu'au sommet `"L"`. Pourquoi la modification de la clé `"vu"` est-elle indispensable ?

* *
*