
TP #15 — Graphes #2 : Parcours

Lecture/écriture de graphes. Si vous le souhaitez, vous pouvez (notamment lors des tests !) utiliser les fonction de lecture et écriture de graphe du [TP13](#). Si besoin, une proposition de correction de ces fonctions est disponible [ici](#).

Exercice 1.

Bicoloriage

On considère des graphes connexes, non orientés (sans boucle), *bipartis*, représentés par liste de listes d'adjacences, où les sommets sont numérotés par des entiers consécutifs depuis 0.

Un exemple de tel graphe, donné dans le format texte du TP13 est :

UNDIRECTED

S = 10

0 -- 1

0 -- 3

2 -- 1

2 -- 9

4 -- 1

4 -- 3

6 -- 7

6 -- 1

6 -- 5

8 -- 3

8 -- 7

8 -- 9

L'objectif de cet exercice est d'implémenter une fonction de bicoloriage, qui pour un graphe en entrée tel que ci-dessus utilise un parcours afin d'associer une couleur rouge ou noir à chacun des sommets, de sorte qu'aucun sommet noir (resp. rouge) ne soit adjacent à un sommet noir (resp. rouge).

Autrement dit, on cherche à mettre en évidence la structure bipartie du graphe via un coloriage.

1. Écrivez une fonction `bicolour(G)` qui utilise un parcours quelconque pour bicolorier G , et renvoie une paire de dictionnaires (*red*, *black*) des sommets coloriés en rouge et en noir respectivement.

Si G n'est pas biparti et connexe, votre fonction pourra échouer arbitrairement (par exemple en ne fournissant pas un coloriage valide).

Conseils. Si vous le souhaitez (mais ce n'est pas obligatoire), vous pouvez prendre comme point de départ l'algorithme générique de parcours décrit en

cours (en le débarrassant éventuellement de ce qui est ici inutile).

2. Testez.

Exercice 2.

Détection de cycle

On considère des graphes non orientés et sans boucles, représentés par liste de listes d'adjacence, où les sommets sont numérotés par des entiers consécutifs depuis 0.

L'objectif de cet exercice est d'implémenter une fonction qui, pour un graphe en entrée tel que ci-dessus, utilise un parcours pour compter son nombre de composantes connexes et en déduire s'il est acyclique. Dans le cas d'un graphe cyclique, on ne cherchera pas à renvoyer un cycle, mais on est tout de même intéressé par le nombre de composantes connexes visitées.

Comment procéder

Il suffit de maintenir une structure permettant de déterminer globalement quels sommets du graphe n'ont pas été encore visités et, tant qu'un tel sommet existe, effectuer un parcours depuis ce sommet pour visiter ceux de sa composante connexe ; chaque itération de cette boucle externe augmente donc de un le nombre de composantes connexes.

Puisque chaque sommet devra ultimement être visité, un choix possible & efficace pour cette structure globale est simplement une `list` (un tableau) de la même longueur que le nombre de sommets, initialisée à `False`. On partage cette structure pour tous les parcours (qui lorsqu'ils visitent un sommet marquent simplement à `True` l'entrée correspondante), et on maintient une variable pointant vers le dernier sommet depuis lequel un parcours a été effectué. Chaque fois qu'un parcours a été terminé, il suffit d'*avancer* ce pointeur à la prochaine valeur à `False` dans la structure globale ; s'il n'y en a pas, cela veut dire que l'on a visité l'ensemble des sommets du graphe et qu'on peut donc s'arrêter.

1. Écrivez une fonction `acyclic(G)` qui implémente l'algorithme ci-dessus, et qui pour un graphe $G = (S, A)$ renvoie un quadruplet $(\#S, \#A, \#C, a)$, où $\#C$ est le nombre de composantes connexes de G , et a vaut `True` si G est acyclique, et `False` sinon.
2. Testez.
3. Analysez précisément le coût de votre fonction. S'il n'est pas linéaire en la taille de son entrée, c'est que vous vous êtes mal débrouillé-e et vous devez recommencer.

Exercice 3.

Saute-canton du misanthrope (un exercice de J.-B. Bianquis)

Le but de cet exercice est de calculer plusieurs types de plus-court-chemins à travers les communes de France.

1. Téléchargez les fichiers `communes.csv` et `adjacences.csv`.

- `communes.csv` contient, pour chaque commune, un identifiant entier unique, le code INSEE (identifiant alphanumérique unique), le nom, le département et la population ;
 - `adjacence.csv` contient la liste des communes immédiatement adjacentes (l'identifiant utilisé est l'identifiant entier unique du fichier `communes.csv`). Chaque paire de communes adjacentes n'est présente qu'une seule fois (s'il y a une ligne pour x, y , la ligne y, x n'est pas présente).
2. Écrivez une fonction `lire_communes()` qui construit et renvoie une `list com` telle que `com[i]` contient un tuple représentant les informations de la commune d'identifiant unique i . On fera bien attention au fait que l'information de population doit être stockée comme un entier. (Au besoin, allez revoir le sujet du TP13 pour une documentation succincte sur la lecture de fichiers textes en Python. Pour convertir une chaîne de caractères `s` « écrivant » un entier en l'entier correspondant, vous pouvez utiliser `int(s)`.) (Vous pouvez dans cette fonction « coder en dur » l'emplacement où vous avez sauvegardé le fichier `communes.csv`.)
 3. Testez.
 4. Écrivez une fonction `construire_graphe()` qui construit et renvoie le graphe non-orienté et non pondéré représentant les liens d'adjacence entre communes. Le graphe renvoyé devra être représenté par liste de listes d'adjacences. (Si vous trouvez cela plus pratique, vous pouvez ajouter des arguments à `construire_graphe`.)
 5. Testez.

Le jeu [Saute canton](#) consiste à partir d'une commune aléatoire et à passer de commune adjacente en commune adjacente en essayant d'arriver le plus rapidement possible à une commune d'au moins 50 000 habitants.

6. Écrivez une fonction `saute_canton(com, adj, u)` qui renvoie un chemin de longueur minimale reliant la commune passée en argument à une commune (quelconque) d'au moins 50 000 habitants. Vous êtes libre de choisir la représentation du chemin renvoyé (par exemple une liste du nom des communes).

Comment procéder. Un simple parcours en largeur convient ici, puisqu'on cherche à minimiser le nombre de sauts (et pas par exemple la distance à vol d'oiseau). Vous pouvez essayer d'implémenter ce parcours (ainsi que la reconstruction de chemin) par vous-même (en utilisant par exemple une `deque` de `collections` pour la file), ou bien adapter le parcours générique du cours.

Bien entendu, il ne sert à rien de continuer le parcours une fois qu'une solution a été trouvée. Si aucune solution n'est trouvée (ce qui est possible), vous pouvez vous contenter de renvoyer une liste vide.

7. Testez (par exemple en jouant à *saute canton*, mais soyez raisonnable !)
8. Déterminez la (ou l'une des) commune la plus «perdue» de France suivant le critère de saute canton (celle pour laquelle le chemin minimal vers une «grande» commune est le plus long possible).

Remarque. On trouve un chemin de longueur 32. Cependant, si vous êtes à jour des dernières (?) fusions de commune vous constaterez que le fichier utilisé dans ce TP n'est plus correct, de même que la solution trouvée :(

On s'intéresse désormais au cas d'un voyageur misanthrope : il souhaite voyager d'une commune *A* à une commune *B* (toutes deux fixées), mais tient absolument à rencontrer le moins de personnes possible en route. Autrement dit, il cherche un chemin minimisant la somme des populations des communes traversées.

9. Expliquez comment construire un graphe permettant de résoudre ce problème à l'aide de l'algorithme de Dijkstra.
10. Écrivez une fonction `construire_graphe2(com)` effectuant cette construction
11. Écrivez une fonction `saute_canton_misanthrope(com, adj, u, v)` permettant de résoudre ce problème.
Si vous le souhaitez, vous pouvez utiliser l'implémentation de file de priorité `PriorityQueue` du module Python `queue` (notamment les fonctions `PriorityQueue`, `pq.empty`, `pq.put`, `pq.get`). N'hésitez pas à demander de l'aide si nécessaire.
12. Quel chemin conseillez-vous au misanthrope pour relier Villeurbanne à La Mulatière ? Montrouge à Aubervilliers ?

Remarque. Les chemins les plus courts sont respectivement Villeurbanne — Lyon — La Mulatière et Montrouge — Paris — Aubervilliers, mais notre ami misanthrope est prêt à de très longs détours (on trouve des chemins de longueur 22 et 67, respectivement).