/tikz/,/tikz/graphs/
conversions/canvas coordinate/.code=1 , conversions/coordinate/.code=1

Sujet n°7

Algorithmique & Structures de données

Les graphes [1]

`Temps de réalisation: 3h00mn`

L'objectif du TP est de :
- définir des structures de données pour représenter un graphe
- implémenter les opérations élémentaires sur les graphes
- résoudre quelques problèmes classiques de graphes

Échelle de progression :

● « débutant·e » ■ ● « confirmé·e » ■ ● « avancé·e » ■ ● « expert·e » ■

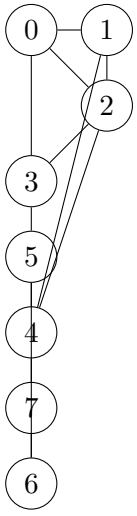
Ex.	01	02	03	04	05	06	07
	●	●	●	●	●	●	●

Pour acquérir un niveau de compétence donné, il faut achever **tous** les exercices de ce niveau et des niveaux inférieurs. L'objectif est d'atteindre le niveau *confirmé·e*(●)*à l'issue du TP*.

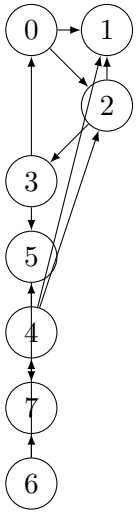
Table des matières

Introduction

Un graphe non orienté G est la donnée d'un ensemble de sommets V (pour *Vertices*) et d'un ensemble d'arêtes $E \subseteq \{\{x, y\} : x, y \in V\}$ (pour *Edges*) liant deux sommets entre eux. Deux sommets



(a) Graphe non orienté



(b) Graphe orienté

FIGURE 1 – Exemples de graphes simples.

liés par une arête sont dits *adjacents*—ou voisins. Un exemple de graphe simple [2] non orienté G est donné sur la Figure 1a avec $V = \{0, 1, 2, 3, 4, 5, 6, 7\}$ et $E = \{\{0,1\}, \{0,2\}, \{0,3\}, \{1,2\}, \{1,4\}, \{2,3\}, \{2,4\}, \{3,5\}, \{4,5\}, \{4,6\}, \{5,7\}, \{6,7\}\}$.

Un graphe orienté $\overrightarrow{G}$ est la donnée d'un ensemble de sommets V et d'un ensemble d'arcs $E \subseteq V \times V$ liant un sommet à un autre. Si (u, v) est un arc, u est un *prédécesseur* de v, et v est un *successeur* de u. La Figure 1b montre un exemple de graphe orienté $\overrightarrow{G}$.

1 Vers un type `Graph`

Exercice 1: Les représentations d'un graphe

On décrit la structure d'un graphe principalement à l'aide de ses **listes d'adjacence**, ou d'une **matrice d'adjacence** [3].

On rappelle que la matrice d'adjacence Adj d'un graphe orienté $G = (V, E)$ est une matrice carrée de taille $|V| \times |V|$, telle que :

$$\mathsf{Adj}_{i,j} = \begin{cases} 1 & \text{si } (i,j) \in E \\ 0 & \text{sinon} \end{cases}$$

On définit de la même manière la matrice d'adjacence d'un graphe non orienté. Notez que celle-ci est alors symétrique.

Les listes d'adjacence d'un graphe non orienté—resp. orienté—$G = (V, E)$ correspondent aux $|V|$ listes des voisins—resp. successeurs—de chaque sommet.

Parmi les représentations alternatives, on trouve la matrice d'incidence, le *line graph*, ou encore l'ensemble des cliques maximales. En pratique, la matrice d'adjacence et les listes d'adjacence sont les représentations les plus usitées car elles offrent de bonnes performances dans la plupart des résolutions de problèmes sur le graphes. Le choix des listes d'adjacence ou de la matrice d'adjacence est fonction de la *densité* du graphe : un graphe creux sera avantageusement représenté par des listes d'adjacence, tandis qu'un graphe dense se prête mieux à une représentation sous forme matricielle.

1. d'après un sujet de Guillaume Raschia
2. graphe sans arête réflexive ni arête dédoublée.
3. dont une extension est la matrice Laplacienne, obtenue par conjonction avec la matrice des degrés du graphe.

Question 1. Donner la complexité en espace de la représentation d'un graphe par :

1. matrice d'adjacence ;
2. listes d'adjacences.

Question 2. Donner, pour chacune des 2 représentations—listes ou matrice d'adjacence—, la complexité en temps de :

1. la recherche des voisins ;
2. l'ajout d'arête ;
3. la suppression d'arête ;
4. le test d'adjacence entre 2 sommets.

Question 3. Écrire la fonction

```
def am_is_undirected(adj_m: AdjMatrix) -> bool
```

qui accepte une matrice d'adjacence `adj_m` et décide s'il peut ou non s'agir d'un graphe non orienté. On supposera que la matrice d'adjacence est bien formée, ie. elle est carrée et n'admet que des valeurs parmi $\{0, 1\}$.

En outre, nous utilisons l'alias `AdjMatrix` pour désigner simplement le type des listes de listes de nombres entiers, là où elles sont employées comme des matrices d'adjacence :

```
AdjMatrix = List[List[int]]
```

Question 4. Proposer, de manière similaire, l'alias de type `AdjLists` pour désigner le type des listes d'adjacence d'un graphe. Nous considérons des graphes dont les sommets sont repérés par—et limités à–leurs clés, ie. des chaînes de caractères.

Un graphe pondéré—orienté ou non—, aussi appelé graphe valué, est la donnée d'un triplet $G = (V, E, \nu)$ dans lequel ν est une fonction de valuation qui affecte un poids à chaque arc—ou arête—du graphe. Sur la Figure 2, l'arête $\{4, 6\}$ du graphe pondéré a la valeur 5 : $\nu(\{4, 6\}) = 5$. On supposera dans la suite que les poids sont des valeurs entières strictement positives.

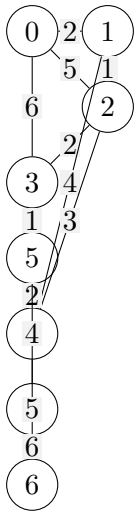


Figure 2 – Exemple de graphe valué.

Question 1. Quel est l'impact de la valuation d'arcs/arêtes sur les types `AdjMatrix`, `AdjLists` et sur la fonction `am_is_undirected` ? Modifier le code en conséquence.

Désormais toutes les structures de données et les algorithmes proposés doivent accepter des graphes simples, orientés ou non, pondérés ou non.

Question 2. Écrire la fonction

```
def al_is_undirected(adj_l: AdjLists) -> bool
```

qui accepte les listes d'adjacence d'un graphe et décide s'il peut ou non s'agir d'un graphe non orienté.

Question 3. Écrire la fonction

```
def al_undirect(adj_l: AdjLists) -> None
```

qui complète les listes d'adjacence `adj_l` pour transformer le graphe orienté en graphe non orienté.

Question 4. Écrire les fonctions de conversion suivantes :

```
def am_to_al(adj_m: AdjMatrix, vertices: Sequence[str] = ()) -> AdjLists
def al_to_am(adj_l: AdjLists) -> AdjMatrix
```

qui transforment une matrice d'adjacence en listes d'adjacence et vice versa. Le paramètre `vertices` fournit l'ensemble des sommets, utile à la construction des listes d'adjacence. En l'absence de ce paramètre, les n sommets sont numérotés de 1 à n.

Exercice 2: Vers une bibliothèque de manipulation de graphes

Une fois familiarisés avec les différentes représentations des graphes, il convient de créer une structure de données pour permettre leur instanciation et quelques opérations élémentaires.

Question 1. Dans un module dédié, définir une structure de données `Graph` à même de représenter un graphe simple orienté ou non, pondéré ou non, sous la forme de ses listes d'adjacence. Le type `Graph` ainsi créé retiendra également s'il est orienté ou non, s'il est pondéré ou non, son ordre et sa taille.

Question 2. Dans un second temps, il est nécessaire de se doter d'opérations de création et d'édition du graphe. Proposer les fonctions suivantes :
- ajouter et supprimer un sommet ;
- ajouter/mettre à jour et supprimer un arc/arête pondérée ou non ;
- créer un graphe simple non pondéré, orienté ou non, à partir de la liste de ses arcs/arêtes et éventuellement une liste complémentaire de ses sommets.

Un effet secondaire accepté—souhaité ?—de l'ajout d'arc/arête consiste à ajouter les sommets reliés qui ne sont pas déjà dans l'ensemble des sommets du graphe. De façon duale, la suppression d'un sommet entraîne mécaniquement la suppression des arcs/arêtes dont il est une extrêmité.

Question 3. Ajouter à la bibliothèque de graphes les fonctions suivantes :
- donner le voisinage d'un sommet ;
- calculer le degré d'un sommet ;
- calculer le degré maximal du graphe ;
- calculer la densité du graphe.

Question 4. Proposer une opération de contraction d'un graphe. La contraction supprime un arc/arête d'un graphe en fusionnant ses deux extrémités. Autrement dit, la contraction G/e d'un arc $e = (x, y)$ au sommet x rend le sommet x adjacent à tous les voisins précédents de y. Le sommet y est ainsi supprimé, au profit du sommet x. Par convention, s'il existe dans G deux arcs (x, z) et (y, z), alors le poids de l'arc (x, z) après contraction $G/(x, y)$ est le minimum des deux poids originaux.

Exercice 3: Génération de graphes

Dans cet exercice, nous considérons le type générique `Graph` défini à l'exercice précédent.

Question 1. Proposer une fonction de transformation d'un graphe orienté en un graphe non orienté.

Question 2. Écrire une fonction qui transforme un graphe non orienté en graphe orienté, de façon parfaitement aléatoire : chaque arête $\{a, b\}$ du graphe est transformée indifféremment en arc (a, b) ou (b, a). Tous les sommets du graphe original sont conservés.

Question 3. Écrire une fonction qui, à partir d'un ensemble de n sommets et un réel $p \in [0, 1]$, renvoie un graphe aléatoire ayant pour toute paire de sommets u, v, l'arc (u, v) avec probabilité p. Quel graphe obtient-on avec $p = 0$? $p = 1$?

2 Algorithmique de graphe

Exercice 4: Parcours de graphe

Un parcours de graphe est une façon de visiter successivement les sommets d'un graphe à partir d'un sommet initial, en progressant de proche en proche dans le graphe. C'est un outil fréquemment employé pour étudier des propriétés globales telles que la connexité, le plus court chemin, l'arbre couvrant, *etc.*

En préambule, on appelle *bordure* $\Gamma(S)$ d'une partie $S \subseteq V$ dans un graphe $G = (V, E)$, le sous-ensemble des sommets de $V \setminus S$ qui sont les extrémités d'un arc/arête dont l'origine est dans S.

Un parcours de G depuis un sommet v est alors une séquence ℓ de sommets de G telle que :

— chaque sommet de G apparaît une fois et une seule dans ℓ ;

— chaque sommet de ℓ—sauf le premier—appartient à la bordure du sous-ensemble des sommets placés avant lui dans ℓ.

Il existe principalement deux paradigmes pour parcourir un graphe : en largeur et en profondeur. Le parcours en largeur consiste, à partir d'un sommet, à visiter tous les voisins, puis à reprendre le parcours à partir de chaque voisin pour visiter la « deuxième couronne », et ainsi de suite. Le parcours en profondeur opère différemment, en suivant intégralement un chemin jusqu'à un cul-de-sac ou une boucle, puis en empruntant successivement les chemins de plus grand préfixe commun jusqu'à épuiser tous les sommets accessibles du graphe.

Par exemple, à partir du graphe représenté sur la Figure 3 et de son sommet 0, le parcours en largeur donne la séquence $(0, 1, 2, 3, 4, 5, 6, 7)$, tandis que le parcours en profondeur produit la séquence $(0, 1, 5, 7, 2, 3, 4, 6)$.

Dans les questions suivantes, vous utiliserez de nouveau la structure `Graph` précédemment définie.

Question 1. Écrire la fonction

```
def bfs_explore(g: Graph, init: str) -> Iterator[str]
```

qui parcourt le graphe en largeur (*breadth-first search*) à partir d'un sommet initial `init`. Seuls les sommets accessibles à partir de `init` sont visités. À chaque appel, la fonction retourne le sommet suivant dans l'ordre *bfs* : c'est un *générateur*[4].

Remarque: *L'implémentation récursive de l'algorithme est très intuitive. L'implémentation itérative, quant à elle, exploite une file contenant les sommets à traiter.*

Question 2. En suivant une démarche analogue, programmer et appliquer l'algorithme `dfs_explore` de parcours de graphe en profondeur (`dfs` pour *depth-first search*).

4. Voir le Sujet n°4 pour un exemple de générateur.

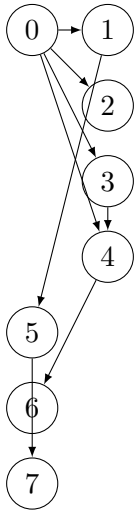


FIGURE 3 – Exemple de graphe à parcourir.

Remarque: *L'implémentation itérative de l'algorithme exploite une pile contenant les sommets à traiter.*

Exercice 5: Connexité

Un graphe non orienté $G = (V, E)$ est dit *connexe* si toute paire de sommets x et y dans V est reliée par une chaîne. Un sous-graphe induit connexe maximal d'un graphe orienté est appelé une *composante connexe*. Une composante connexe $\mathcal{C} = (V_\mathcal{C}, E_\mathcal{C})$ d'un graphe $G = (V, E)$ est donc entièrement décrite par le sous-ensemble $V_\mathcal{C} \subseteq V$ de ses sommets ; ses arêtes $E_\mathcal{C} \subseteq E$ étant toutes les arêtes de G dont les extrêmités sont dans $V_\mathcal{C}$.

Par exemple, le graphe présenté sur la Figure 4 se compose de 2 composantes connexes : $\{0, 1, 2, 3, 4, 6\}$ et $\{5, 7, 8\}$, qui déterminent de façon univoque les 2 sous-graphes induits visuellement séparés sur la Figure 4.

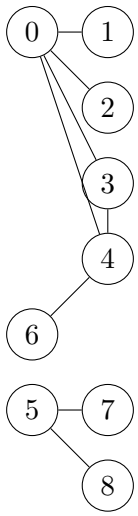


FIGURE 4 – Graphe composé de deux composantes connexes.

QUESTION 1. Écrire une fonction qui, étant donné un graphe non orienté, partitionne l'ensemble des sommets suivant ses composantes connexes. Chaque composante connexe sera représentée par « une couleur » $i \in \mathbb{N}$.

Par analogie, un graphe orienté $G = (V, E)$ est *fortement connexe* si, pour toute paire x et y de sommets de V, il existe un chemin $x \to_G^* y$ et un chemin $y \to_G^* x$. Une *composante fortement connexe* est alors un sous-graphe induit maximal fortement connexe.

QUESTION 2. Écrire une fonction qui colorie les composantes fortement connexes d'un graphe orienté.

> **Remarque:** *Il existe principalement 3 méthodes astucieuses pour résoudre ce problème : l'algorithme de* KOSARAJU*, celui de* TARJAN *et un algorithme à base de chemins.*

3 Pour aller plus loin...

Exercice 6: Arbre couvrant de poids minimum

L'algorithme de KRUSKAL est un algorithme de recherche d'arbre couvrant de poids minimum dans un graphe simple connexe non orienté et valué.

QUESTION 1. Programmer et appliquer la procédure de KRUSKAL pour déterminer un arbre couvrant de poids minimum à partir d'un graphe et étant donnée une racine, ie. le sommet initial.

Exercice 7: Plus court chemin

QUESTION 1. En quoi consiste l'algorithme de DIJKSTRA ?

QUESTION 2. Rappeler les conditions d'application de l'algorithme.

QUESTION 3. Programmer et appliquer l'algorithme de DIJKSTRA pour déterminer la distance—longueur du plus court chemin—d'un sommet v donné à chacun des sommets d'un graphe.