TP 8

Graphes

I - Graphes

I.1 - Définitions

- Un **graphe non orienté** est la donnée d'un couple (S, A), S représentant un ensemble fini de **sommets** et A un ensemble fini d'**arêtes** reliant deux sommets.
  On dit que le sommet s_2 est un **voisin** du sommet s_1 si il existe une arête reliant ce deux sommets.
- Un **graphe orienté** est la donnée d'un couple (S, A), S représentant un ensemble fini de **sommets** et A un ensemble fini d'**arcs** (des flèches) reliant deux sommets.
  On dit que le sommet s_2 est un **voisin** du sommet s_1 si il existe un arc partant de s_1 vers s_2.
- On appelle **chemin** (ou chaîne) reliant deux sommets A et B toute suite d'arêtes (d'arcs) consécutives reliant A à B.
- Un sommet s_2 est dit **accessible** depuis un sommet s_1 lorsqu'il existe au moins un chemin reliant s_1 à s_2.
- On dit qu'un graphe est **connexe** si chaque sommet de ce graphe peut être relié par au moins un chemin à n'importe quel autre sommet.

Exemples :

- Le graphe des utilisateurs de Facebook est un graphe non orienté : chaque sommet représente un utilisateur et il y a une arête entre deux sommets lorsque deux utilisateurs sont amis.
- Le graphe des utilisateurs d'Instagram est un graphe orienté : chaque sommet représente un utilisateur et il y a un arc de x vers y lorsque x est un follower de y.
- Un réseau de bus est un graphe non orienté : chaque sommet représente une station de bus et les arêtes représentent les liaisons entre ces stations.

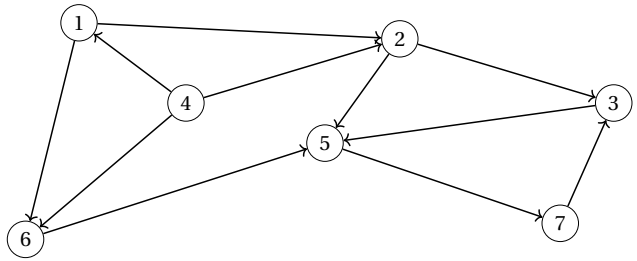
I.2 - Représentation

On peut représenter un graphe $G = (S, A)$ par :

- une **matrice d'adjacence**.
  Si n est le nombre de sommets du graphe G, la matrice d'adjacence de G est une matrice carrée de taille n dont le coefficient de la ligne i et de la colonne j vaut 1 s'il existe une arc allant du sommet i au sommet j et 0 sinon.
- une **liste d'adjacence**.
  Cette liste contient pour chaque sommet s de S, la liste de ses voisins.
  L'utilisation d'un dictionnaire en Python est particulièrement adapté pour cette représentation. On parle alors de dictionnaire d'adjacence.

Exercice 1 (*Matrice et liste d'adjacence*)

Voici un exemple de graphe orienté avec lequel nous travaillerons.
On peut lui associé un graphe non orienté en remplaçant les arcs par des arêtes.



1. (a) Écrire à la main la matrice d'adjacence et la liste d'adjacence de ce graphe orienté.
   (b) Écrire à la main la matrice d'adjacence et la liste d'adjacence du graphe non orienté associé.
2. Écrire une fonction `Dico_vers_Mat(D)` qui prend en paramètre le dictionnaire d'adjacence d'un graphe et renvoie sa matrice d'adjacence.
3. Écrire une fonction `Mat_vers_Dico(M)` qui prend en paramètre la matrice d'adjacence d'un graphe et renvoie son dictionnaire d'adjacence.
4. Tester les fonctions précédentes sur le graphe de l'exercice 1.

Exercice 2

1. On considère un graphe orienté G donné par son dictionnaire d'adjacence.
   (a) Écrire une fonction `DegreOut(G,x)` qui prend comme arguments un graphe `G` et un sommet `x`, et qui renvoie le degré sortant de `x`, c'est à dire le nombre d'arcs dont ce sommet est une origine.
   (b) Écrire une fonction `DegreIn(G,x)` qui prend comme arguments un graphe `G` et un sommet `x`, et qui renvoie le degré entrant de `x`, c'est à dire le nombre d'arcs dont ce sommet est une cible.
   (c) Écrire une fonction `Arcs(G)` qui prend comme argument un graphe `G` et qui renvoie le nombre d'arcs de ce graphe.
   (d) Modifier cette dernière fonction pour l'adapter au cas de graphes non orientés.
2. On considère un graphe orienté G donné par sa matrice d'adjacence.
   (a) Écrire une fonction `DegreOut2(M,x)` qui prend comme arguments la matrice `M` d'un graphe et un sommet `x`, et qui renvoie le degré sortant de `x`, c'est à dire le nombre d'arcs dont ce sommet est une origine.
   (b) Écrire une fonction `DegreIn2(M,x)` qui prend comme arguments la matrice `M` d'un graphe et un sommet `x`, et qui renvoie le degré entrant de `x`, c'est à dire le nombre d'arcs dont ce sommet est une cible.

Exercice 3

On considère un graphe orienté G donné par son dictionnaire d'adjacence.

1. Écrire une fonction `AjoutSommet(G,x)` qui prend comme arguments un graphe `G` et un sommet `x` et qui ajoute ce sommet (isolé) au graphe si celui n'est pas déjà présent.
2. Écrire une fonction `AjoutArc(G,x,y)` qui prend comme arguments un graphe `G` et deux sommets `x` et `y`, et qui ajoute un arc allant de `x` vers `y` si cet arc n'existe pas déjà.
3. Écrire une fonction `SupprimeSommet(G,x)` qui prend comme arguments un graphe `G` et un sommet `x` et qui supprime ce sommet ainsi que tous les arcs ayant ce sommet comme origine ou comme extrémité.

II - Parcours de graphe

II.1 - Pile et file

- **Pile.** Une pile est un collection d'objet pour laquelle les opérations d'ajout et de suppression d'un élément se font à partir de la même extrémité (appelée sommet de la pile).
  Le principe des opérations repose sur la règle LIFO : « Last In, First Out ».
  Les listes Python permettent d'implémenter une pile.
  - ⋆ **Empiler :** La méthode `append()` permet d'ajouter un élément sur le sommet de la pile (le dernier élément de la liste).
  - ⋆ **Dépiler :** La méthode `pop()` permet de supprimer un élément sur le sommet de la pile (le dernier élément de la liste).
- **File.** Une file est un collection d'objet pour laquelle les opérations d'ajout et de suppression d'un éléments se font à partir de deux extrémités : on ajoute un élément en fin de liste et on supprime un élément en début de liste.
  Le principe des opérations repose sur la règle FIFO : « First In, First Out ».
  Les listes Python permettent d'implémenter une file.
  - ⋆ **Emfiler :** La méthode `append()` permet d'ajouter un élément à la fin de la file (le dernier élément de la liste).
  - ⋆ **Défiler :** La méthode `pop(0)` permet de supprimer un élément au début de la file (le premier élément de la liste).

  Cependant, l'implémentation d'une file par une liste n'est pas optimal car l'opération de "dépilage" est de complexité $O(n)$.
  Il existe un autre type de données qui permet des ajouts et suppressions rapides (en $O(1)$). Il s'agit du sous-module `deque` (double ended queue) du module `collections`.
  Nous utiliserons ce type pour implémenter les piles et les files en Python.

```
from collections import deque

f  = deque()     # Création une file/pile vide
f.append(a)      # Ajoute l'élément a en sommet de pile (en fin de file)
x = f.pop()      # Enlève l'élément présent sur le sommet de la pile
x = f.popleft() # Enlève l'élément présent au début de la file
n = len(f)       # Nombre d'éléments de f
```

II.2 - Parcours en profondeur

- Objectif.
  On cherche à lister l'ensemble des sommets accessibles depuis un sommet de départ.
  On utilise une pile stocker les sommets à traiter et une liste pour stocker les sommets accessibles. Cette liste donnera à la fin la liste des sommets accessibles depuis le sommet de départ.
  Le règle LIFO donne alors un parcours en profondeur (on va le plus loin possible en partant du sommet de départ).
- Algorithme.
  - ⋆ Placer le sommet de départ dans la pile des sommets à traiter.
  - ⋆ Tant que la pile des sommets à traiter n'est pas vide :
    - – Dépiler le sommet S présent au sommet de la pile et si il n'a pas encore été noté "accessible", l'ajouter à la liste des sommets accessibles.
    - – Empiler tous les voisins de S (qui deviennent donc des sommets à traiter).

Exercice 4

On considère le graphe de l'exercice 1.
Appliquer sur ce graphe l'algorithme de parcours en profondeur en détaillant l'évolution de la pile et de la liste de sommets visités à chaque itération.

II.3 - Parcours en largeur

- Objectif.
  On cherche à lister l'ensemble des sommets accessibles depuis un sommet de départ.
  On utilise une file stocker les sommets à traiter et une liste pour stocker les sommets déjà visités (donc accessibles). Cette liste donnera à la fin la liste des sommets accessibles depuis le sommet de départ.
  Le règle FIFO donne alors un parcours en largeur (on explore le graphe en "cercles concentriques" à partir du sommet de départ).
- Algorithme.
  - ⋆ Placer le sommet de départ dans la file des sommets à traiter et le marquer comme accessible.
  - ⋆ Tant que la file des sommets à traiter n'est pas vide :
    - – Défiler le premier sommet de la file.
    - – Enfiler tous ses voisins non encore notés "accessibles" et les marquer comme accessibles. Ces voisins deviennent donc des sommets à traiter.

Exercice 5

On considère le graphe de l'exercice 1.
Appliquer sur ce graphe l'algorithme de parcours en largeur en détaillant l'évolution de la pile et de la liste de sommets visités à chaque itération.

III - Implémentation sous Python

Exercice 6 (*Parcours en profondeur*)

1. Écrire une fonction `Parcours_Profondeur(Graphe,Depart)` qui prend comme argument un graphe (donné par son dictionnaire d'adjacence) ainsi qu'un sommet Depart et qui renvoie la liste de tous les sommets accessibles depuis le sommet Depart par un parcours en profondeur.
2. Tester cette fonction sur le graphe de l'exercice 1.

Exercice 7 (*Parcours en largeur*)

1. Écrire une fonction `Parcours_Largeur(Graphe,Depart)` qui prend comme argument un graphe (donné par son dictionnaire d'adjacence) ainsi qu'un sommet Depart et qui renvoie la liste de tous les sommets accessibles depuis le sommet Depart par un parcours en largeur.
2. Écrire une fonction `Parcours_Largeur_2(Graphe,Depart)` qui fait le même travail et qui indique en plus la plus petite distance au départ des sommets accessibles.
   On pourra remplacer la liste des sommets accessibles par un dictionnaire dont les clés sont les sommets et les valeurs sont la distance au départ.
3. Tester les fonctions précédentes sur le graphe de l'exercice 1.

Exercice 8 (*Applications*)

1. Écrire une fonction `Chemin(Graphe,x,y)` qui prend comme argument un graphe(donné par son dictionnaire d'adjacence) et deux sommets `x`, `y` et qui renvoie True si il existe un chemin entre `x` et `y`.
2. Écrire une fonction `Connexe(Graphe)` qui prend comme argument un graphe non orienté (donné par son dictionnaire d'adjacence) et renvoie True si il est connexe.
3. Tester les fonctions précédentes sur le graphe de l'exercice 1.