Les graphes

Table des matières

Ce qu'il faut savoir faire à l'issue du chapitre :

- .
- .

1 Les graphes

1.1 Introduction

Imaginons le mini réseau social suivant :

Cécilia est amie avec Mathieu et Brice

Mathieu est ami avec Cécilia, Brice, Franck et Nathan

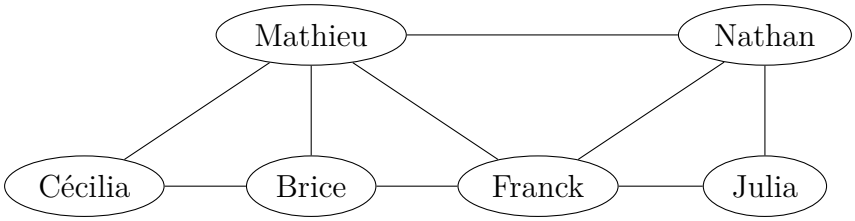
Brice est ami avec Cécilia, Mathieu, Franck

Franck est ami avec Brice, Mathieu, Nathan, Julia

Julia est amie avec Franck et Nathan.

Nathan est ami avec Mathieu, Julia et Franck

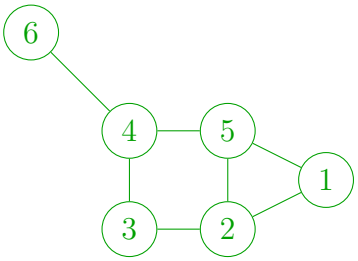
La description de ce réseau social, malgré son faible nombre d'abonnés, est déjà quelque peu rébarbative, alors imaginez cette même description avec un réseau social comportant des millions d'abonnés !
Il existe un moyen plus "visuel" pour représenter ce réseau social : les **graphes** :



1.2 Définition

Définition 10.1

Un **graphe simple** est un graphe composé de sommets et d'arêtes :

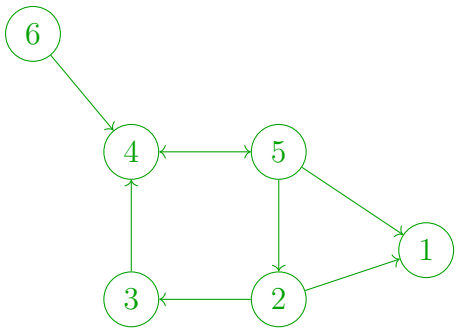


Remarque

Le terme **arête** désigne donc la relation entre deux sommets dans le cas d'un graphe non orienté.

Définition 10.2

Un **graphe orienté** est un graphe avec des flèches appelées **arcs**.

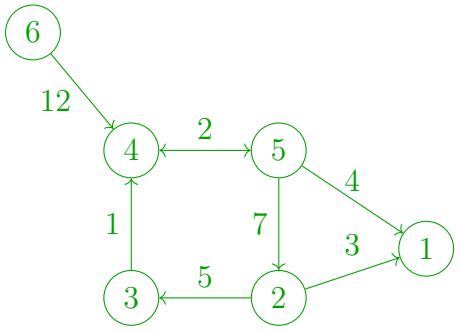


Remarque

Le terme **arc** désigne donc la relation entre deux sommets dans le cas d'un graphe orienté.

Définition 10.3

Dans certains cas, toutes les arêtes (resp les arcs) ne se valent pas, et on peut ainsi attribuer un poids à chaque arêtes (resp chaque arc). On parle de **graphe pondéré** (resp. de **graphe orienté pondéré**) :



Définition 10.4

Une **chaine** est une suite d'arêtes consécutives dans un graphe non orienté, un peu comme si on se promenait sur le graphe. On la désigne par les lettres des sommets qu'elle comporte.

Exemple

Définition 10.5

Un **chemin** est une suite d'arcs consécutifs dans un graphe orienté, un peu comme si on se promenait sur le graphe. On la désigne par les lettres des sommets qu'elle comporte.

Exemple

Définition 10.6

Un **cycle** est une chaine ou un chemin qui commence et se termine au même sommet.

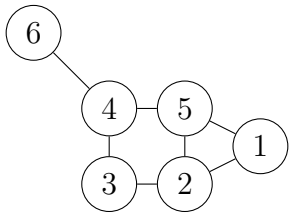
Exemple

2 Implémentation d'un graphe

2.1 Implémentation d'un graphe à l'aide d'une matrice d'adjacence

2.1.1 Graphe simple

Reprenons le graphe simple pour commencer :



La matrice associée au graphe ci-dessus est :

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

Cette matrice est obtenue en remplissant un tableau où chaque ligne correspond au sommet de départ et chaque colonne au sommet d'arrivée :

		A	B	C	D	E	F
		❶	❷	❸	❹	❺	❻
A	❶	0	1	0	0	1	0
B	❷	1	0	1	0	1	0
C	❸	0	1	0	1	0	0
D	❹	0	0	1	0	1	1
E	❺	1	1	0	1	0	0
F	❻	0	0	0	1	0	0

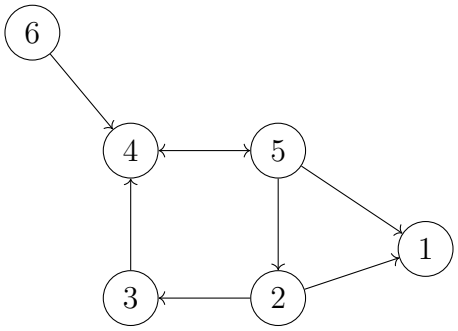
Il faut savoir qu'à chaque ligne correspond un sommet du graphe et qu'à chaque colonne correspond aussi un sommet du graphe. À chaque intersection ligne i-colonne j (ligne i correspond au sommet i et colonne j correspond au sommet j), on place un 1 s'il existe une arête entre le sommet i et le sommet j, et un zéro s'il n'existe pas d'arête entre le sommet i et le sommet j.

La case verte indique qu'il n'y a pas de relation de D vers B.

On remarque ici une symétrie par rapport à une des diagonale ; ceci s'explique par la fait d'avoir un graphe simple.

2.1.2 Graphe orienté

Reprenons le graphe orienté :

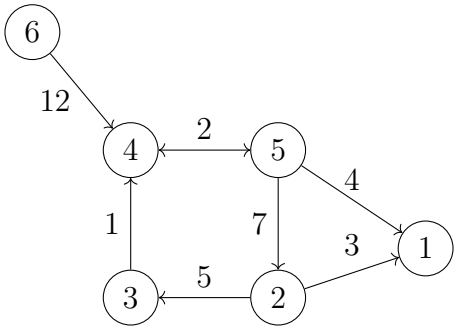


La matrice associée au graphe ci-dessus est :

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

2.1.3 Graphe pondéré

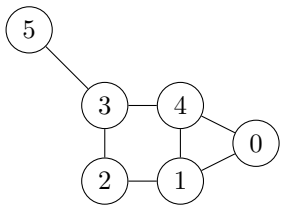
Reprenons le graphe pondéré et orienté :



Il suffit ici de remplacer les "1" par les pondérations respectives.
La matrice associée au graphe ci-dessus est :

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 3 & 0 & 5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 0 \\ 4 & 7 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 12 & 0 & 0 \end{pmatrix}$$

Exercice 10.1
On considère le graphe suivant :



1. Construire la fonction **matrice_vide(n)** qui prend en argument la taille de la matrice et qui renvoie la matrice carrée n*n avec False pour chaque coefficient

```
>>> matrice_vide(6)
[[False, False, False, False, False, False], [False, False, False, False, False, False
], [False, False, False, False, False, False], [False, False, False, False, False, Fal
se], [False, False, False, False, False, False], [False, False, False, False, False, F
alse]]
```

2. Construire la fonction **ajouter_arc(mat,s1,s2)** de paramètres mat (une matrice), s1 et s2 des sommets. La fonction crée dans la matrice un lien qui matérialise l'arc orienté de s1 vers s2
3. Créer la matrice ci-dessus.
4. Créer une fonction **afficher_arc** de paramètres mat, s1 et s2 et qui renvoie True si il y un arc orienté de s1 vers s2, False sinon
5. Construire une fonction **sommet_voisinage(mat,s)** de paramètre une matrice mat et un sommet s. La fonction renvoie un tableau avec la liste des sommets voisins de s

```
>>> voisinage_sommet(mat,4)
[0, 1, 3]
>>> voisinage_sommet(mat,2)
[1, 3]
```

Exercice 10.2

Implémenter une classe Graphe avec l'interface ci-dessous :

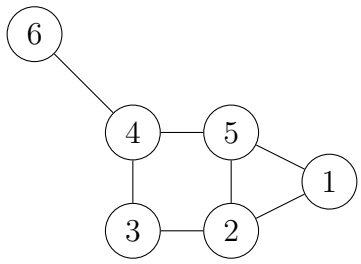
- Un constructeur **___init___(self,n)** avec n la taille de la matrice carrée. Cette classe contiendra deux attributs : n qui sera la taille de la matrice et adj qui sera la matrice de taille n*n , avec chaque coefficient égal à False
- Une méthode **creer_arc(self,s1,s2)** avec s1 et s2 deux sommets. La fonction indique True dans la matrice pour matérialiser le lien de s1 vers s2
- La méthode **afficher_arc(self,s1,s2)** qui renvoie True s'il y a un lien de s1 vers s2, False sinon.
- La méthode **sommet_voisin(self,s)** de paramètre s un sommet, et qui renvoie la liste des sommets voisins à s.

Créer l'objet t qui correspond à l'arbre ci-dessus.
**

2.2 Implémentation avec des listes adjacentes

2.2.1 Cas d'un graphe non orienté

Reprenons le graphe non orienté :

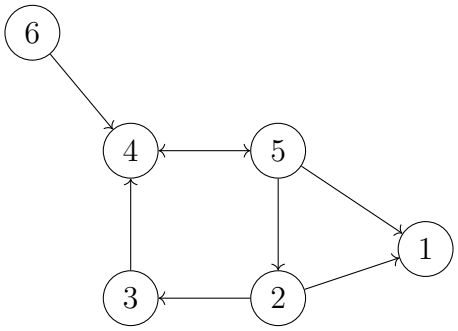


On définit une liste des sommets du graphe. À chaque élément de cette liste, on associe une autre liste qui contient les sommets lié à cet élément :

Sommet	Sommet(s) liés
1	2 ;5
2	1 ;3 ;5
3	2 ;4
4	3 ;5 ;6
5	1 ;2 ;4
6	4

2.2.2 Cas d'un graphe orienté

Il est alors nécessaire ici de faire deux tableaux : un pour les prédécesseurs et un pour les successeurs de chaque sommet :



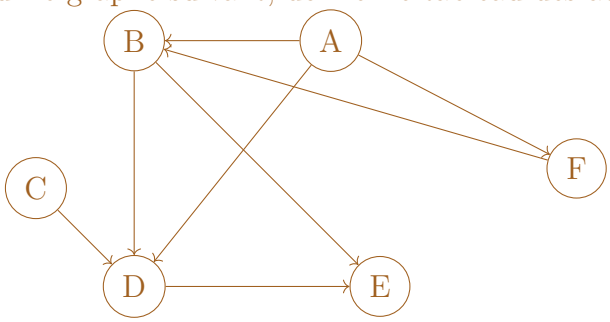
Sommet	Sommet(s) adjacent(s) prédécesseur(s)
1	2 ;5
2	5
3	2
4	3 ;5 ;6
5	4
6	

Sommet	Sommet(s) adjacent(s) successeurs(s)
1	
2	1 ;3
3	4
4	5
5	1 ;2
6	4

Savoir-Faire 10.1

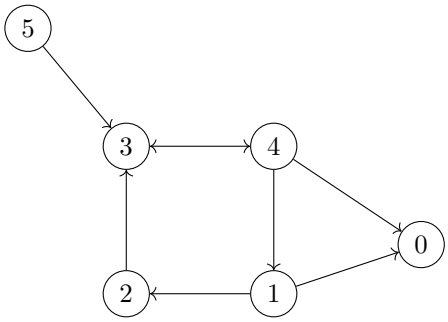
Savoir écrire la liste des adjacences

Pour le graphe suivant, donner le tableau des adjacences.



Exercice 10.3

Considérons le graphe orienté :



On décide de représenter ce graphe par une liste des successeurs.
Réaliser une classe GraphV qui prend comme argument une liste de successeurs. Cette classe comportera une méthode **est_lie(self,i,j)** qui renvoie True s'il y a un lien de i vers j, False sinon. La classe comportera aussi une méthode **graph_vers_matrice(self)** qui renvoie la matrice d'adjacence au graphe.

```
>>> g = GraphV([[],[0,2],[3],[4],[1,0],[3]])
>>> g.est_lie(5,3)
True
>>> g.est_lie(3,5)
False
>>> g.est_lie(3,3)
False
>>> g.graphe_vers_matrice()
[[0, 0, 0, 0, 0, 0], [1, 0, 1, 0, 0, 0], [0, 0, 0, 1, 0, 0], [0, 0, 0, 0, 1, 0], [1, 1,
0, 0, 0, 0], [0, 0, 0, 1, 0, 0]]
```

***

Remarque

Comment choisir l'implémentation à utiliser (matrice d'adjacence ou liste d'adjacence) ?

le choix se fait en fonction de la densité du graphe, c'est-à-dire du rapport entre le nombre d'arêtes et le nombre de sommets. Pour un graphe dense on utilisera plutôt une matrice d'adjacence.
certains algorithmes travaillent plutôt avec les listes d'adjacences alors que d'autres travaillent plutôt avec les matrices d'adjacences. Le choix doit donc aussi dépendre des algorithmes utilisés.

Exercice 10.4

Créer une classe **GrapheNO(self,n)** où n est le nombre de sommets du graphe. On utiliser une matrice d'adjacence. On implémentera les méthodes **ajouter_arete(self,s1,s2)**, **arete(self,s1,s2)**, **voisins(self,s)**, **sommet(self)** et **retirer_arete(self,s1,s2)**.

Exercice 10.5

Dessiner tous les graphes non orientés ayant 3 sommets

Exercice 10.6

Noé, Isa, Karim, Lola, Véro et Fred doivent suivrent des cours de Maths(M), PC,SVT, NSI et Français (F).
Tous ne doivent pas suivrent tous les cours, car certains ont déjà validé des matières.
Le tableau suivant donne les matières que chaque élèves doit suivre :

Nom	Noé	Isa	Karim	Lola	Véro	Fred
Matières	M-PC-SVT	PC	M-SVT	PC-SVT-NSI	NSI-F	SVT-F

L'objectif est de prévoir un planning optimal pour que chacun puisse suivre les cours. Est-il possible de mettre Maths et PC en même temps par exemple ? (non, car Noé doit suivre M et PC)

1. Construire un graphe non orienté à 5 sommets (M-PP-SVT-NSI-F) en considérant qu'une arête entre deux sommets signifie une incompatibilité . Par exemple, on reliera le sommet M et le sommet PC, car il y a une incompatibilité entre M et PC (pour Noé par exemple, qui doit suivre les deux).
2. On coloriera chaque sommet avec un minimum de couleur et en respectant :
   - Chaque sommet doit avoir une couleur
   - Chaque sommet doit avoir une couleur différente de son voisin.
3. En déduire un planning optimal possible.
   ***

☛ Le problème de la coloration d'un graphe avec un minimum de couleur est un exercice difficile . Il n'existe pas, pour l'instant, d'algorithme efficace pour répondre à cette question.

Exercice 10.7

Comprendre et analyser ces fonctions, qui permettent de colorier un graphe.

```
def coloriage(g):
    """ coloriage d'un graphe avec un algorithme glouton"""
    couleur = {} # dico avec les couleurs numérotées avec des ent à partir de zéro
                 # couleur= {2:3,3:4, 4:4} signifie que les sommet 1 est colorié en coul 3 et abs
                 # sommet 3 et sommet 4 en coul 4
    n = 0
    for s in g.sommets():
        c = mex(g.voisins(s),couleur)  # c est la couleur pour le sommet s en cours
        couleur[s] = c
        n = max(c+1,n) #explications plus tard
    return couleur, n

def mex(voisins,couleur):
    '''voisins est une liste de voisins, couleur un dico
    renvoie la plus petite couleur non utilisée'''
    n = len(voisins)
    dispo = [True] * (n + 1)
    for v in voisins :
```

```
            if v in couleur and couleur[v] <= n :
                dispo[couleur[v]] = False
        for c in range(n+1):
            if dispo[c]:
                return c
```

3 Algorithme sur les graphes

Il existe 2 méthodes pour parcourir un graphe :

- Le parcours en largeur d'abord
- Le parcours en profondeur d'abord

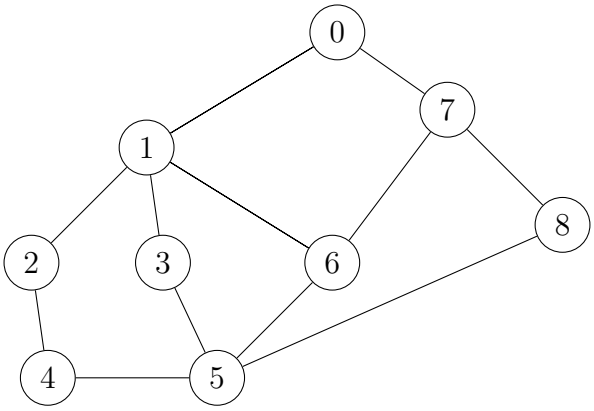
3.1 Le parcours en largeur d'abord

Nous allons travailler sur un graphe G(V,E) avec V l'ensemble des sommets de ce graphe et E l'ensemble des arêtes de ce graphe.

On adoptera un code couleur :

- Sommet de couleur verte si le sommet n'a pas encore été visité
- Sommet de couleur rouge si le sommet a été visité

```
1  VARIABLE
2  G : un graphe
3  s : sommet(origine)
4  u : sommet
5  v : sommet
6  f : file (initialement vide)
7  # On part du principe que pour tout sommet est initialement vert
8  Fonction PARCOURS_ LARGEUR(G,s)
9      s.couleur ← rouge
10     enfiler (s,f)
11     TANT QUE f non vide FAIRE
12         u ← defiler(f)
13         POUR chaque sommet v adjacent au sommet u FAIRE
14             SI v.couleur n'est pas rouge ALORS
15                 v.couleur ← rouge
16                 enfiler(v,f)
```



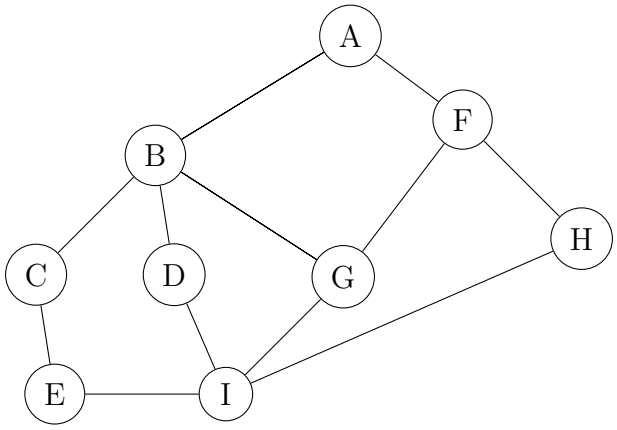
Remarque
- On commence à visiter A

Exercice 10.8
Utiliser l'algorithme du parcours en largeur d'abord pour ce graphe, et donner l'ordre des sommets visités.
***

Exercice 10.9
1. Créer une classe file, avec les méthode enfiler, défiler et est_vide.
2. Reprendre la classe GrapheNO
3. Implémenter cet algorithme (on pourra insérer un print au moment où le sommet passe au rouge)

3.2 le parcours en profondeur d'abord

```
1  VARIABLE
2  G : un graphe
3  u : noeud
4  v : noeud
5  f : file (initialement vide)
6  # On part du principe que pour tout sommet est initialement vert
7  DEBUT
8  Function PARCOURS_ PROFONDEUR(G,u)
9      u.couleur ← rouge
10     for chaque sommet v adjacent à u do
11         if v couleur n'est pas rouge then
12             PARCOURS_ PROFONDEUR(G,v)
13         end
14     end
15 end
```



Utiliser l'algorithme du parcours en profondeur d'abord pour ce graphe, et donner l'ordre des sommets visités, en sachant qu'il n'y a pas qu'une seule solution.

3.3 Recherche de chaînes ou de chemins

3.4 Recherche de cycles

4 Implémentation en python

Il est possible de travailler avec des listes d'adjacences en Python en utilisant les dictionnaires :

```
#liste d'ajacence pour le graphe cartographie
l = {'1':('2','5'), '2':('1', '3', '5'), '3':('2','4'),
 '4':('3', '5','6'), '5':('1', '2', '4'), '6':('4')}
```