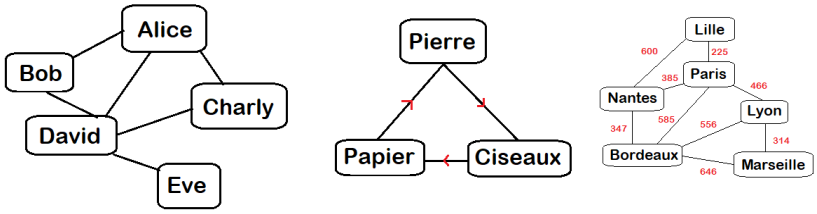
1 Définitions

1.1 Idée générale et exemples

Différentes situations illustrées par des graphes :



1. Graphe des amis (graphe non-orienté) ;
2. Règles du pierre-papier-ciseaux (graphe orienté) ;
3. Routes et distances entre des villes (graphe pondéré).

1.2 Définitions mathématiques

Graphe orienté : Un graphe orienté G est la donnée de deux ensembles :

1. V un ensemble fini de *sommets* (Vertices) ;
2. $E \subseteq V \times V$ un ensemble d'*arêtes* (Edges). $E \subseteq V \times V$ signifie que E consiste en un ensemble de couples (v_1, v_2) avec v_1, $v_2 \in V$.

On interprète ces ensembles de la manière suivante : les sommets sont des objets reliés entre eux par les arêtes, arêtes pouvant n'aller que dans un sens. Plus précisément, deux sommets v_1 et v_2 sont reliés si et seulement si le couple (v_1, v_2) est présent dans E.

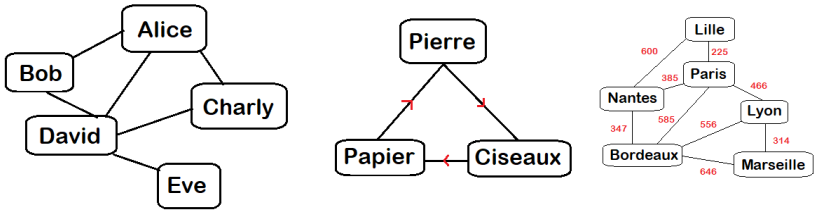
Graphe non-orienté : Un graphe non orienté est un graphe orienté dans lequel chaque arête est présente dans les deux sens :

$$(v_1, v_2) \in E \Rightarrow (v_2, v_1) \in E$$

Graphe pondéré : Un graphe pondéré est un graphe orienté muni d'une *fonction de poids* $w : E \longrightarrow \mathbb{R}$ (weight) associant à chaque arête un certain poids.

1.3 Mathématisation des exemples

Reprenons nos exemples :



1. — $V = \{\text{Alice}, \text{Bob}, \text{Charly}, \text{David}, \text{Eve}\}$
   — $E = \{$(Alice,Bob) ,(Bob,Alice) ,(Alice,David) ,(David,Elice) ,(Alice,Charly) ,(Charly,Alice) ,(Bob,David) ,(David,Bob) ,(Charly,David) ,(David,Charly) ,(David,Eve) ,(Eve,David)$\}$
2. — $V = \{\text{Pierre}, \text{Papier}, \text{Ciseaux}\}$
   — $E = \{$(Pierre, Ciseaux), (Ciseaux, Papier), (Papier, Pierre)$\}$
3. — $V = \{$ Lille, Paris, Nantes, Bordeaux, Lyon, Marseille $\}$
   — $E =$ long...
   — Table de w :

(v_1, v_2)	$w(v_1, v_2) = w(v_2, v_1)$
(Lille, Paris)	225
(Lille, Nantes)	600
(Paris, Nantes)	385
(Paris, Lyon)	466
(Paris, Bordeaux)	585
(Nantes, Bordeaux)	347
(Lyon, Bordeaux)	556
(Lyon, Marseille)	314
(Marseille, Bordeaux)	646

1.4 Exercices

Pour chacune des situations suivantes, déterminer le type de type de graphe le plus adapté à la modélisation et proposer un exemple (avec au moins 4 sommets) sous la forme d'un schéma puis détailler la formalisation mathématique :

1. lieux d'un jeu vidéo ;
2. "followers" sur un réseau social
3. liens internes d'un site web ;
4. trafic routier.

2 Représentation par listes d'adjacence

Une première possibilité d'implémentation d'un graphe consiste à donner, pour chaque sommet, la liste des sommets étant reliés (de manière orientée ou non) à celui-ci.

2.1 Un exemple d'implémentation à la main

Par exemple, pour l'exemple 1, une implémentation peut être :

```
noms = ["Alice", "Bob", "Charly", "David", "Eve"]
#ici "Alice" sera le sommet 0, "Bob" le sommet 1, etc.

amis = [[]]*5
amis[0] = [1,2,3] #alice a pour amis Bob et Charly
amis[1] = [0,3] #etc.
amis[2] = [0,3]
amis[3] = [0,1,2,4]
amis[4] = [3]
```

Ici on définit une correspondance entre des indices et les noms des personnes représentées dans le tableau `noms` puis, pour chaque indice, on donne la liste des indices des amis correspondants dans le tableau `amis`.

Ou bien, en programmation orientée objet :

```
class Personne:
    def __init__(self, nom_personne):
        self.nom = nom_personne
        self.amis = []

a = Personne("Alice")
b = Personne("Bob")
c = Personne("Charly")
d = Personne("David")
e = Personne("Eve")

a.amis += [b, c, d]
b.amis += [a, d]
c.amis += [a, d]
d.amis += [a, b, c, e]
e.amis += [d]
```

Ici, chaque personne est définie par son nom et la liste de ses amis (initialement vide).

2.2 Exercices

1. Dans les deux implémentations précédentes, écrire :
   (a) une fonction / une méthode affichant les noms des amis d'une personne donnée.

(b) une fonction permettant de savoir si deux personnes sont amies ou non.

2. Représenter avec l'implémentation de votre choix l'exemple 2.

2.3 Liste d'adjacence : le cas des graphes pondérés

Si on veut représenter un graphe pondéré, il faut en plus de la donnée des sommets et des arêtes, renseigner le poids de chaque arête. Une possibilité consiste à modifier les listes d'adjacences de sorte que celles-ci contiennent des tuples (voisin, poids de l'arête associée).

2.4 Amélioration : classe générale par dictionnaire d'adjacence

```
#programme 37
class Graphe:
    def __init__(self):
        """cree un graphe vide"""
        self.adj = {}

    def ajouter_sommet(self, s):
        """ajoute un sommet s"""
        if not s in self.adj:
            self.adj[s] = set()

    def ajouter_arc(self, s1, s2):
        """cree un arc de s1 vers s2"""
        self.ajouter_sommet(s1)
        self.ajouter_sommet(s2)
        self.adj[s1].add(s2)

    def arc(self, s1, s2):
        return s2 in self.adj[s1]

    def sommets(self):
        return list(self.adj)

    def voisins(self, s):
        return self.adj[s]
```

3 Représentation par matrice d'adjacence

3.1 Un exemple à la main

Soit $V = \{0, 1, ..., n-1\}$ un ensemble de sommets numérotés et $G = (V, E)$ un graphe non-pondéré. Une autre possibilité de représentation est la matrice d'adjacence M de G. Celle-ci est de taille $n \times n$ et définie par :

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

Par exemple, dans l'exemple 1, en remplaçant Alice, Bob, etc. par 0, 1, 2, 3, 4, on peut donner la matrice suivante :

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

L'implémentation est assez proche de la version tableau des listes d'adjacence, en reprenant l'exemple 1 :

```
noms = ["Alice", "Bob", "Charly", "David", "Eve"]
#ici "Alice" sera le sommet 0, "Bob" le sommet 1, etc.

amis = 5*[[]]#
amis[0] = [0,1,1,1,0] #alice a pour amis Bob et Charly
amis[1] = [1,0,0,1,0] #etc.
amis[2] = [1,0,0,1,0]
amis[3] = [1,1,1,0,1]
amis[4] = [0,0,0,1,0]
```

3.2 Amélioration : classe générale par dictionnaire d'adjacence

```python
#programme 36
class Graphe:
    def __init__(self, n):
        """cree un graphe de taille n sans arc"""
        self.n = n
        self.adj = [[False]*n for i in range(n)]

    def ajouter_arc(self, s1, s2):
        """cree un arc de s1 vers s2"""
        self.adj[s1][s2] = True

    def arc(self, s1, s2):
        return self.adj[s1][s2]

    def voisins(self, s):
        return [i for i in range (self.n) if self.adj[s][i]]
```

3.3 Exercices

1. Implémenter l'exemple 2 (pierre, papier, ciseaux) avec la classe générale.
2. Écrire une fonction affichant "gagné", "perdu" ou "égalité" étant donné deux choix au pierre, papier, ciseaux et utilisant la matrice d'adjacence.

Comparaison des deux représentations

On choisira une représentation ou l'autre en fonction de la situation à représenter. Quelques pistes :

— La représentation matricielle est plus gourmande en espace mémoire car pour un graphe à n sommets, la matrice est toujours de taille n^2 alors qu'avec les listes d'adjacence, on n'a que $|E| \leq n^2$ voisins à retenir.

— En revanche, savoir si une arête est présente entre deux sommets est plus rapide dans la représentation matricielle, car elle ne nécessite qu'un appel de $M[i][j]$ alors que dans la représentation par listes d'adjacence une recherche dans une liste (de complexité linéaire en sa taille) est nécessaire.

— On privilégiera donc une représentation matricielle dans les cas où le nombre de voisins par sommet est important (matrice dite dense, beaucoup de 1) et la représentation par liste d'adjacence si le nombre de voisin par sommet est petit (matrice dite vide).

— Enfin, on verra plus tard que selon la représentation utilisée les algorithmes de graphes associés ne s'effectuent pas de la même manière, ce qui peut également orienter le choix de l'implémentation.

4 Vocabulaire

En plus des sommets et des arrêtes, on définit les notions suivantes sur les graphes :

— Deux sommets sont dits **adjacents** s'il existe une arrête entre les deux. Le **voisinage** d'un sommet est l'ensemble des sommets lui étant adjacents. Dans le cas des graphes orientés, on ne considère que les sommets accessibles à partir de celui-ci.

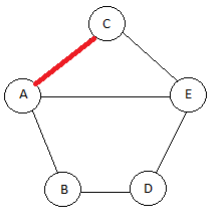


FIGURE 1 – A et B sont adjacents

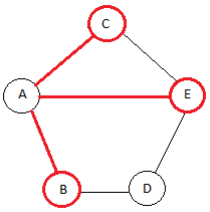


FIGURE 2 – Le voisinage de A

— Étant donnée deux sommets A et B, un **chemin de A à B** est une suite finie de sommets adjacents deux à deux et menant de A à B. Formellement $[S_1, ..., S_n]$ chemin de A à B si :
  1. $S1 = A$, $S_n = B$
  2. $\forall i \in \{1, ..., n-1\}$, $(S_i, S_{i+1}) \in E$

— un chemin ne passant jamais deux fois par la même arrête est dit **simple**. Un chemin ne passant jamais deux fois par le même sommet est dit **élémentaire**. Un chemin simple reliant un sommet à lui même est appelé un **cycle**.

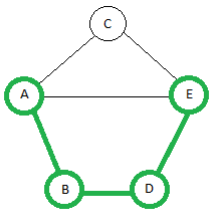


FIGURE 3 – Un chemin de A à E

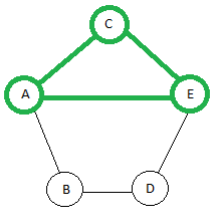


FIGURE 4 – Un cycle

— La **longueur d'un chemin** est son nombre d'arêtes. Dans le cas des graphes pondérés, la somme des poids de ses arêtes est appelé son **coût**.
— La **distance entre deux sommets** est la plus petite longueur des chemins les reliant s'il y en a. S'il n'en n'existe pas, la distance n'est alors pas définie (mais on la définit parfois comme égale à $+\infty$).

Remarque : Dans l'exemple précédent, le chemin considéré est de longueur 3 mais la distance entre A et E est de 1.
— un graphe non-orienté pour lequel il existe un chemin entre toute paire de sommets est dit **connexe**.
— un graphe orienté pour lequel il existe un chemin entre toute paire de sommets est dit **fortement connexe**.

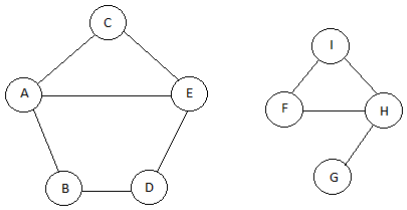


FIGURE 5 – Un graphe non-connexe

5 Exemple d'algorithme : Coloration de graphe

5.1 Problème de coloration de graphe

Étant donné un graphe G, une coloration de G est une assignation de couleur pour chaque sommet de G telle qu'aucune paire de sommets adjacents n'ait la même couleur.

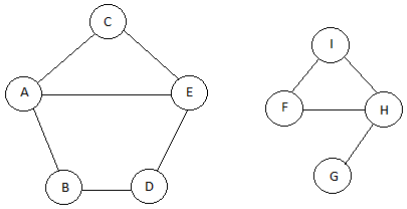


FIGURE 6 – Colorer les graphes avec le moins de couleur possible

Une telle coloration existe toujours (il suffit de prendre une couleur par sommet). Il existe cependant un certain nombre de questions moins évidentes :

1. Peut-on colorer un graphe avec seulement 2 couleurs ? (relativement facile)
2. Peut-on colorer un graphe avec seulement $k \geq 2$ couleurs ? (difficile)
3. Quel est le nombre minimal de couleur utilisable pour colorer un graphe ? (difficile)

Les questions 2 et 3 sont des problèmes difficiles (NP-complets), on ne connaît pas aujourd'hui d'algorithmes de complexité polynomiale y répondant.

Cependant, on peut proposer un algorithme glouton (non-exact) proposant une coloration.

Algorithme 1 : Coloration gloutonne d'un graphe

Données : Un graphe G
Résultat : Une coloration des sommets de G
pour *s sommet de G* **faire**
  assigner à s la plus petite couleur non-utilisée par les voisins de s
fin

Cette formulation très simple appelle bien évidement une implémentation plus précise. en particulier, l'assignation des couleur peut se faire selon un dictionnaire `couleur` associant à chaque sommet du graphe, une couleur représentée par exemple par un entier entre 0 et $n-1$.

5.2 Exercice

1. Écrire une fonction `mex(voisins, couleur)` prenant en argument une liste de voisins (ou ensemble selon la représentation choisie) et un dictionnaire d'assignation de couleurs et renvoyant la plus petite couleur disponible.
2. En déduire une fonction `coloriage(g)` renvoyant un coloriage du graphe `g`.