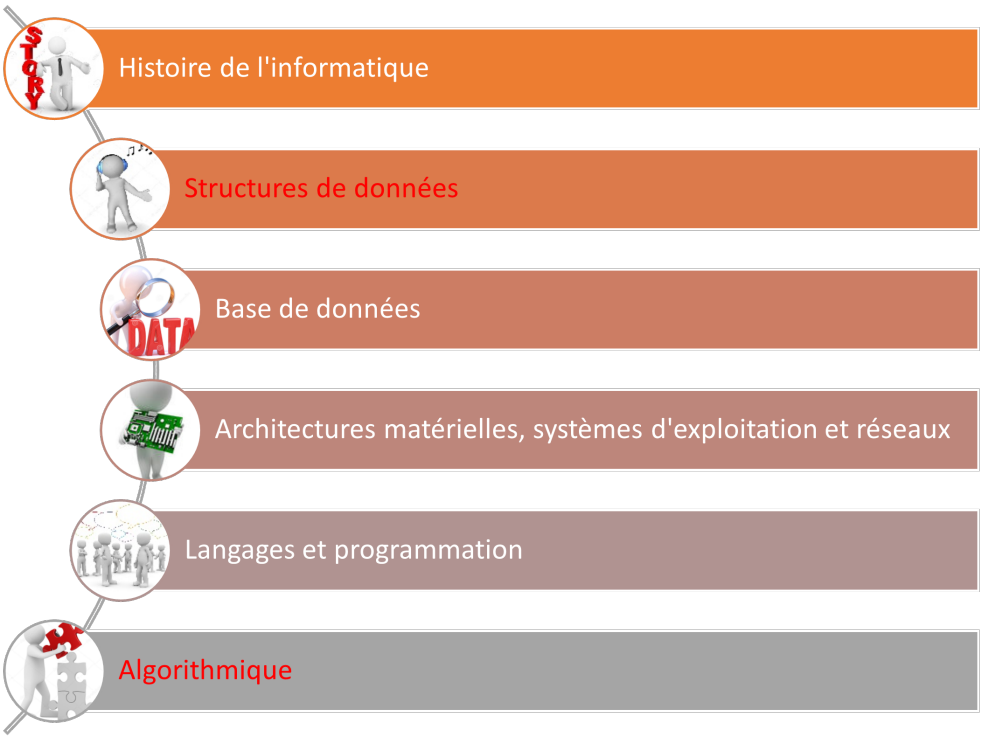


Table des matières

1 Introduction à la théorie des graphes

1.1 Premières définitions

Définition :

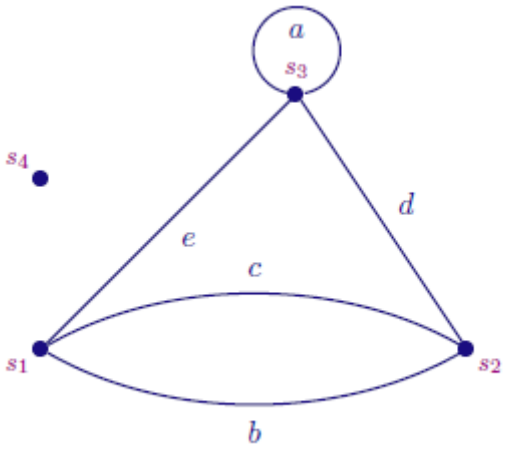
- Un **graphe** G (non orienté) est constitué d'un ensemble S de points appelés **sommets**, et d'un ensemble A **d'arêtes**, tels qu'à chaque arête sont associés deux sommets, appelés ses **extrémités**.
- Deux sommets qui sont les extrémités d'une même arête sont dits **adjacents**.

Remarques :

- Les deux extrémités peuvent être distinctes ou confondues (dans ce dernier cas l'arête s'appelle une boucle).
- Deux arêtes distinctes peuvent avoir les mêmes extrémités (on dit qu'elles sont parallèles).
- La position des sommets et la longueur ou l'allure des arêtes n'ont aucune importance.

Exemple : Nommons G le graphe ci-contre :

- Le sommet s_4 est un sommet isolé.
- L'arête a est une boucle.
- b et c sont des arêtes ayant mêmes extrémités.
- Les sommets s_1 et s_2 sont adjacents, ainsi que s_1 et s_3.

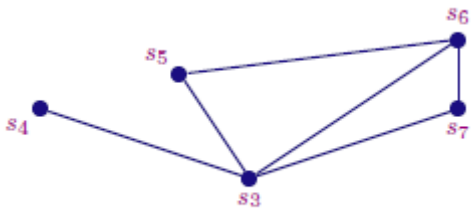


Graphe G_1

Définition :

- **L'ordre** d'un graphe est le nombre de ses sommets.
- On appelle **degré** d'un sommet le nombre d'arêtes dont ce sommet est une extrémité (les boucles étant comptées deux fois).
- Un sommet est pair (respectivement impair) si son degré est un nombre pair (respectivement impair).

Exemple : Donner le degré de chacun des sommets du graphe ci-contre.



Graphe G_2

Définition :
Un graphe est dit **simple** si deux sommets distincts sont joints par au plus une arête et s'il est sans boucle.

Le graphe G_2 ci-dessus est simple.

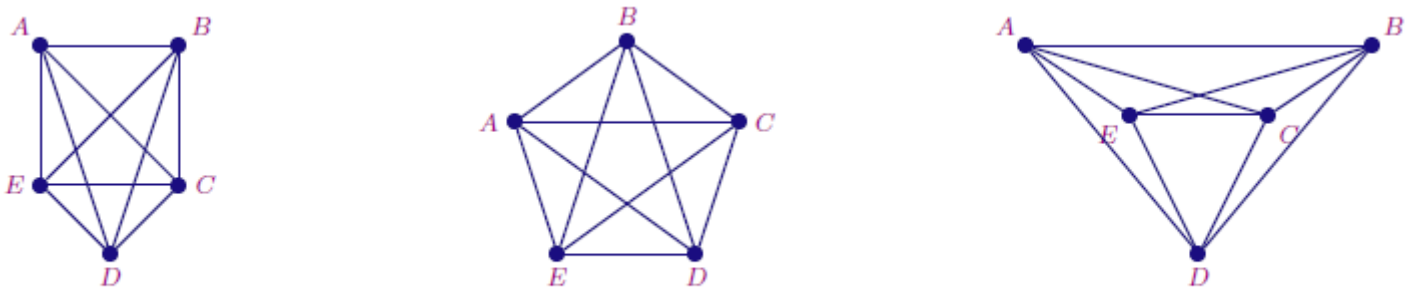
Dans la suite de ce cours, on ne considèrera que des graphes simples.

1.2 Graphes complets.

Définition :
Un graphe simple (sans boucle ni arêtes parallèles) est dit **complet** si tous ses sommets sont adjacents, c'est-à-dire si tous les sommets sont reliés 2 à 2 par une arête et une seule.
On appellera K_n le graphe complet à n sommets.

Dessiner les graphes K_3, K_4 et K_6

Remarque : attention, un graphe est défini par l'ensemble de ses sommets et l'ensemble de ses arêtes, mais sa représentation n'est pas unique :



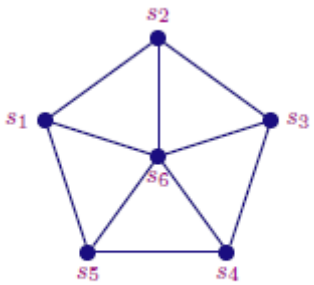
3 représentations du graphe complet K_5

1.3 Chemins et connexité.

Définition :

- Un **chemin** dans un graphe simple G est une suite de sommets $(s_0\,;\,s_1\,;\,s_2\,;\,s_n\,)$.
- La **longueur du chemin** est égale au nombre d'arêtes qui le constituent.
- Le chemin est fermé si $s_0 = s_n$. Si, de plus, toutes ses arêtes sont distinctes, on dit alors que c'est un **cycle**.

Exemple :



Graphe G_3

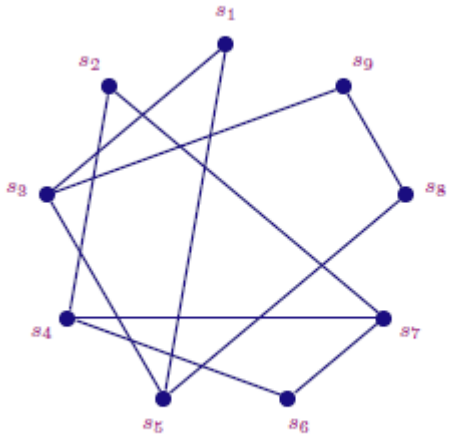
Définition :
Un graphe est **connexe** si deux sommets quelconques sont reliés par un chemin.

Définition :
Soit G un graphe connexe, s et s' deux sommets quelconques de G.
Le graphe étant connexe, il existe au moins une chemin reliant s et s'. On appelle **distance entre s et s'** la plus petite des longueurs des chemins reliant s à s'.

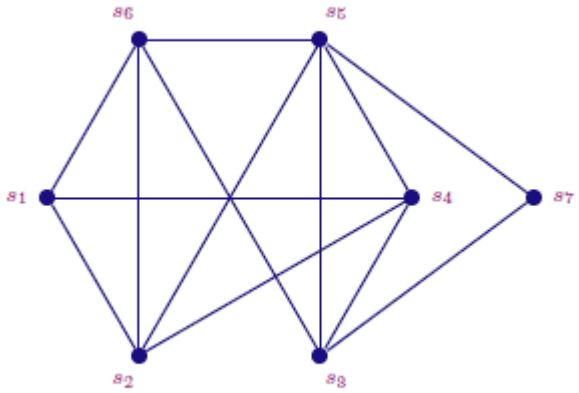
Remarque : si le graphe n'est pas connexe, il existe au moins deux sommets qui ne sont pas reliés par un chemin.

Définition :
On appelle **diamètre d'un graphe connexe**, la plus grande distance entre deux de ses sommets.

Les graphes suivants sont-ils connexes ? Si oui, donner leur diamètre.



Graphe G_4

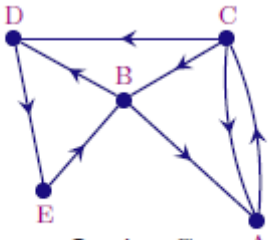


Graphe G_5

1.4 Graphes orientés.

Définition :
On appelle **graphe orienté** un graphe où chaque arête est orientée, c'est-à-dire qu'elle va de l'une de ses extrémités, appelée origine ou extrémité initiale à l'autre, appelée extrémité terminale.

Exemple : Voici un exemple de graphe orienté :



Graphe G_6

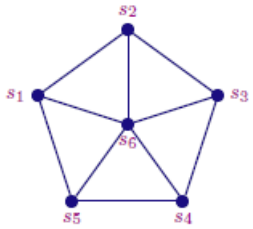
Remarque : Toutes les notions définies précédemment pour un graphe non orienté ont un équivalent pour un graphe orienté, on se contentera d'utiliser le mot orienté pour préciser. En particulier, un chemin orienté est une suite d'arêtes telles que l'extrémité finale de chacune soit l'extrémité initiale de la suivante. On prendra garde au fait que l'on peut définir un chemin (non orienté) sur un graphe orienté. Par exemple, sur un plan de villes où toutes les rues sont en sens unique, un parcours de voiture correspond à un chemin orienté, un parcours piéton correspond à un chemin non orienté.

1.5 Matrice d'adjacence

1.5.1 Cas des graphes non orientés.

Définition :
Soit G un graphe non orienté qui possède n sommets numérotés de 1 à n. On appelle matrice d'adjacence du graphe la matrice $A = (a_{ij})$, où a_{ij} est le nombre d'arêtes joignant le sommet numéro i au sommet numéro j.

Exemple : donner la matrice d'adjacence A du graphe G_3.



Graphe G_3

Remarque : l'allure de la matrice d'adjacence donne des indications sur la nature du graphe. Ainsi :

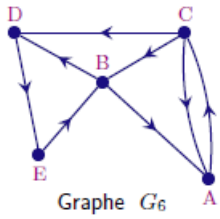
- La matrice d'adjacence d'un graphe sans boucle n'a que des 0 sur la diagonale.
- La matrice d'adjacence d'un graphe sans arête parallèle n'a que des 1 ou des 0.
- La matrice d'adjacence d'un graphe non orienté est symétrique par rapport à sa diagonale.
- La matrice d'adjacence d'un graphe complet n'a que des 1, hormis sur sa diagonale où il y a des 0.

1.5.2 Cas d'un graphe orienté

On a vu que la matrice d'adjacence d'un graphe non orienté est symétrique. Il n'en va pas de même pour celle d'un graphe orienté car si une arête a pour extrémité initiale le sommet A et pour extrémité finale le sommet B, il n'en existe pas forcément une allant de B vers A, aussi dans ce cas le coefficient correspondant aux arêtes allant de A vers B ne sera pas le même que le coefficient correspondant aux arêtes allant de B vers A. On doit donc convenir d'un sens de lecture pour la matrice d'un graphe orienté.

Définition :
Par convention, dans la matrice d'adjacence d'un graphe orienté, le terme a_{ij} désigne le nombre d'arêtes d'origine le sommet i et d'extrémité finale le sommet j.

Exemple : donner la matrice d'adjacence B du graphe G_6.



Graphe G_6

2 Représentation d'un graphe en Python

Il existe de multiples façons de représenter un graphe en machine, selon la nature du graphe, mais aussi selon la nature des opérations à effectuer ou des algorithmes à utiliser sur ce graphe.
Pour toute représentation, on souhaite disposer d'opérations de deux sortes :

- opérations de construction de graphes : obtention de graphe vide, ajout de sommet, ajout d'arête...
- opérations de parcours de graphes (voir paragraphe suivant)

2.1 Matrice d'adjacence

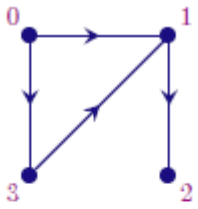
Dans cette première représentation, les sommets du graphe sont supposés être les entiers $0,\ 1, \ldots, n-1$ où n représente le nombre de sommets du graphe. Le graphe est alors représenté par sa matrice d'adjacence `adj`.
On implémente dans la classe `Graphe` ci-dessous cette représentation. Notez que :

- le constructeur de cette classe prend en paramètre le nombre de sommets et alloue la matrice `adj`.
- la méthode `ajouter_arete` permet d'ajouter une arête au graphe.
- la méthode `arete` permet de tester la présence d'une arête entre deux sommets.
- la méthode `voisins` renvoie l'ensemble des sommets adjacents à un sommet donné.

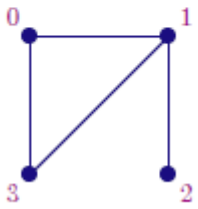
Notez ci-contre les deux graphes g_1 et g_2 (orienté et non orienté) définis comme deux instances de cette classe.

```python
class Graphe:
    """un graphe représenté par une matrice d'adjacence,
       où les sommets sont les entiers 0,1,...,n-1"""

    def __init__(self, n):
        self.n = n
        self.adj = [[0] * n for _ in range(n)]

    def ajouter_arete(self, s1, s2):
        self.adj[s1][s2] = 1

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

    def voisins(self, s):
        v = []
        for i in range(self.n):
            if self.adj[s][i] == 1:
                v.append(i)
        return v

if __name__ == '__main__':
    g1 = Graphe(4)
    g1.ajouter_arete(0, 1)
    g1.ajouter_arete(0, 3)
    g1.ajouter_arete(1, 2)
    g1.ajouter_arete(3, 1)

    print(g1.adj)

    g2 = Graphe(4)
    L = [[0, 1, 0, 1], [1, 0, 1, 1], [0, 1, 0, 0], [1, 1, 0, 0]]
    g2.adj= L
    print(g2.voisins(1))
```

graphe_matrice_adjacence.py



Graphe g_1



Graphe g_2

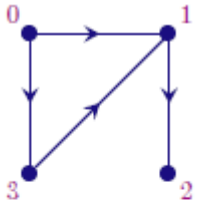
2.2 Dictionnaire d'adjacence

Dans cette nouvelle représentation, on définit un dictionnaire dont les clés sont les sommets du graphe et les valeurs la liste des sommets adjacents à ce sommet. L'ensemble de sommets du graphe est donc l'ensemble des clés du dictionnaire.
Encapsulons dans la classe `Graphe` ci-dessous cette représentation. On dispose :

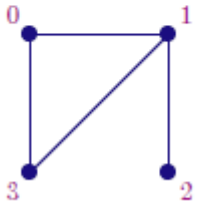
- d'une méthode `ajouter_sommet` pour ajouter un nouveau sommet ;
- d'une méthode `ajouter_arete` pour ajouter une arête au graphe.
- la méthode `arete` permet de tester la présence d'un arc entre deux sommets.
- la méthode `voisins` renvoie l'ensemble des sommets adjacents à un sommet donné.

Comme tout à l'heure, les deux graphes g_1 et g_2 (orienté et non orienté) ci-contre sont définis comme deux instances de cette classe.

```python
class Graphe:
    """un graphe comme un dictionnaire d'adjacence"""

    def __init__(self):
        self.adj = {}

    def ajouter_sommet(self, s):
        if s not in self.adj:
            self.adj[s] = []

    def ajouter_arete(self, s1, s2):
        self.ajouter_sommet(s1)
        self.ajouter_sommet(s2)
        self.adj[s1].append(s2)

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

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

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

if __name__ == '__main__':
    g1 = Graphe()
    g1.ajouter_arete(0, 1)
    g1.ajouter_arete(0, 3)
    g1.ajouter_arete(1, 2)
    g1.ajouter_arete(3, 1)

    print(g1.adj)
    print(g1.sommets())
    print(g1.voisins(0))

    dic = {0: [1, 3], 1: [0, 2, 3], 2: [1], 3: [0, 1]}
    g2 = Graphe()
    g2.adj = dic
    print(g2.adj)
    print(g2.sommets())
    print(g2.voisins(1))
```



Graphe g_1



Graphe g_2

graphe_matrice_adjacence.py

Pour choisir entre matrice d'adjacence ou liste d'adjacence, on peut considérer les éléments suivants :

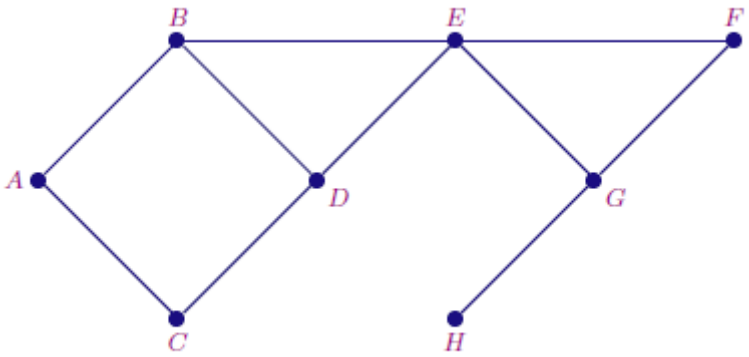
- la densité du graphe : c'est le rapport entre le nombre d'arêtes et le nombre de sommets. Pour un graphe dense on utilisera plutôt une matrice d'adjacence.
- le type d'algorithme que l'on souhaite appliquer. Certains algorithmes travaillent plutôt avec les listes d'adjacences alors que d'autres travaillent plutôt avec les matrices d'adjacences. Le choix doit donc aussi dépendre des algorithmes utilisés

3 Algorithmes sur les graphes

Les algorithmes sur les graphes sont utilisés dans de très nombreux domaines :

- les logiciels de guidage par GPS - la cartographie routière
- les protocoles de routage réseau comme OSPF

Dans ce paragraphe, nous prendrons en exemple le graphe suivant :



Graphe G_7

Sa matrice d'adjacence se définit ainsi :

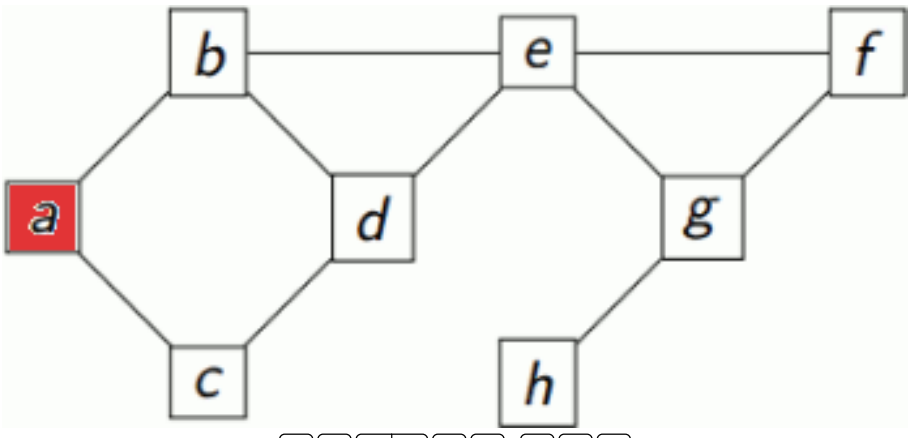
```
matrice = [[0,1,1,0,0,0,0,0],
           [1,0,0,1,1,0,0,0],
           [1,0,0,1,0,0,0,0],
           [0,1,1,0,1,0,0,0],
           [0,1,0,1,0,1,1,0],
           [0,0,0,0,1,0,1,0],
           [0,0,0,0,1,1,0,1],
           [0,0,0,0,0,0,1,0]]
```

3.1 Parcourir un graphe

3.1.1 Parcours en profondeur d'abord (Deep First Search - DFS)

Pour le parcours en profondeur, on commence avec un nœud donné et on explore chaque branche complètement avant de passer à la suivante. Autrement dit, on commence d'abord par aller le plus profond possible. Comme pour les arbres, cet algorithme s'écrit naturellement de manière récursive.
Le principe est le suivant :

1. On choisit un nœud de départ
2. On le marque comme visité
3. Si le nœud est déjà visité, on ne fait rien
4. Sinon, pour chaque nœud connecté, on réitère récursivement cet algorithme



Le code repose sur deux idées :

- l'utilisation d'une liste `vus` pour le marquage de sommets.
- la récursivité pour définir la fonction `parcour_profondeur` qui reçoit en paramètres le graphe `g`, un sommet `s` à partir duquel on commence le parcours et la liste `vus` vide par défaut.

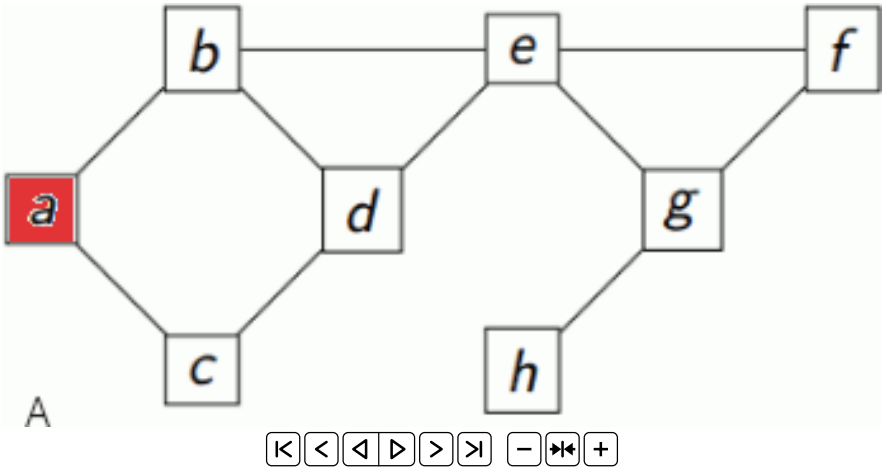
```
def parcours_profondeur(g, s, vus):
    """parcours en profondeur depuis le sommet s"""
    if s not in vus:
        vus.append(s)
        for v in g.voisins(s):
            parcours_profondeur(g, v, vus)
    return vus
```

graphe_DFS.py

3.1.2 Parcours en largeur d'abord (Breadth First Search - BFS)

La encore, on va s'inspirer du parcours en largeur pour les arbres. Pas de fonction récursive ici, mais on va faire appel à une file FIFO pour gérer la liste des nœuds à découvrir. La structure FIFO nous assure que les premiers nœuds découverts seront les premiers explorés, ce qui constitue la différence majeure avec le parcours en profondeur. Le principe de l'algorithme est le suivant :

1. On enfile le nœud de départ.
2. On enfile les nœuds adjacents s'ils ne sont pas déjà présents dans la file et qu'ils n'ont pas déjà été visités.
3. On défile (c'est-à-dire on supprime la tête de file).
4. Tant que la file n'est pas vide, on réitère les points 2 et 3.



Nous utiliserons ici le module `queue` qui offre une structure de file.

```
from queue import *

def parcours_largeur(g, source):
    """parcours en largeur depuis le sommet source"""
    visite = []
    file = Queue()
    file.put(source)
    while file.qsize() > 0:
        sommet = file.get()
        if sommet not in visite:
            visite.append(sommet)
            for s in g.voisins(sommet):
                if s not in visite:
                    file.put(s)
    return visite
```

graphe_BFS.py

On peut aussi penser cet algorithme de parcours en largeur sans file mais en manipulant deux listes. La première `courant` contient tous les sommets situés à une distance `d` du sommet `source`, et la deuxième `suivant` contient les sommets situés à une distance `d + 1` du sommet `source` et que l'on examinera après.
On utilise aussi un dictionnaire du sommet `dist` qui associe à chaque sommet déjà atteint sa distance du sommet `source`.
On procède ainsi :

1. on place le sommet `source` dans `courant` et sa distance est fixée à 0 ;
2. tant que l'ensemble `courant` n'est pas vide :
   (a) on en retire un sommet `s`,
   (b) pour chaque voisin `v` de `s` qui n'est pas encore dans `dist` :
   - on ajoute `v` à l'ensemble `suivant` ;
   - on fixe `dist[v]` à `dist[s] + 1`

   (c) si la liste `courant` est vide, on l'échange avec `suivant`.

```python
def parcours_largeur(g, source):
    """parcours en largeur depuis le sommet source"""
    dist = {source: 0}
    courant = [source]
    suivant = []
    while len(courant) > 0:
        s = courant.pop()
        for v in g.voisins(s):
            if v not in dist:
                suivant.append(v)
                dist[v] = dist[s] + 1
        if len(courant) == 0:
            courant, suivant = suivant, []
    return dist

def distance(g, u, v):
    """distance de u à v (et None si pas de chemin)"""
    dist = parcours_largeur(g, u)
    return dist[v] if v in dist else None
```

graphe_BFS_2.py

3.2 Repérer la présence de cycles

L'algorithme de parcours en profondeur permet de détecter la présence d'un cycle dans un graphe orienté.

En effet dans l'algorithme de parcours en profondeur, on indique dans l'ensemble `vus` l'ensemble des sommets déjà rencontrés, pour autant le fait de "tomber" à nouveau sur un sommet déjà rencontré ne garantit pas que l'on ait un cycle (2 chemins parallèles peuvent mener à un même sommet). Distinguons donc dans l'ensemble `vus` les sommets pour lesquels le parcours en profondeur n'est pas terminé de ceux pour lesquels il l'est. Pour ce faire nous allons utiliser un marquage à trois couleurs :

- blanc pour les sommets non encore atteints ;
- gris pour les sommets déjà atteints dont le parcours en profondeur n'est pas terminé ;
- noir pour les sommets déjà atteints dont le parcours en profondeur est terminé.

Lorsqu'on visite un sommet on procède ainsi :

- s'il est gris, c'est que l'on vient de découvrir un cycle ;
- s'il est noir on ne fait rien ;
- s'il est blanc :
  1. on le colore en gris ;
  2. on visite tous ses voisins récursivement ;
  3. enfin, on le colore en noir.

```python
BLANC, GRIS, NOIR = 1, 2, 3

def parcours_cy(g, couleur, s):
    """parcours en profondeur depuis le sommet s"""
    if couleur[s] == GRIS:
        return True
    if couleur[s] == NOIR:
        return False
    couleur[s] = GRIS
    for v in g.voisins(s):
        if parcours_cy(g, couleur, v):
            return True
    couleur[s] = NOIR
    return False

def cycle(g):
    couleur = {}
    for s in g.sommets():
        couleur[s] = BLANC
    for s in g.sommets():
        if parcours_cy(g, couleur, s):
            return True
    return False
```

 graphe_detction_cycle.py

3.3 Chercher un chemin

La recherche de chemin dans un graphe est quelque chose de très utile : vos logiciels de guidage par GPS utilisent de tels algorithmes pour vous préparer un itinéraire optimisé en distance ou en temps pour aller d'une ville A à une ville B.

On peut à partir de l'algorithme de parcours en profondeur écrire une fonction qui détermine s'il existe un chemine entre deux sommets `u` et `v` du graphe :

```python
def existe_chemin(g, u, v):
    """existe-t-il un chemin de u à v ?"""
    vus = []
    parcours_profondeur(g, u, vus)
    return v in vus
```

 existe_chemin.py

4 Exercices

Exercice n° 1.

1. Dessiner tous les graphes non orientés ayant exactement trois sommets.
2. Sans les dessiner, déterminer le nombre de graphes orientés ayant exactement trois sommets.

Exercice n° 2.

On considère la classe `Graphe` (avec matrice d'adjacence).

1. Ajouter une méthode `afficher`, pour afficher le graphe sous la forme suivante :
   0 − > 1 3
   1 − > 2 3
   2 − > 3
   3 − > 1
2. Ajouter une méthode `degre(s)`, qui renvoie le degré d'un sommet `s` du graphe.
   En déduire une méthode `nb_arete()`, qui renvoie le nombre total d'arêtes du graphe.
3. Ajouter une méthode `supprimer_arete(s1, s2)`, pour supprimer l'arête entre les sommets `s1` et `s2` du graphe. S'il n'y a pas d'arête entre ces deux sommets, la méthode n'a aucun effet.

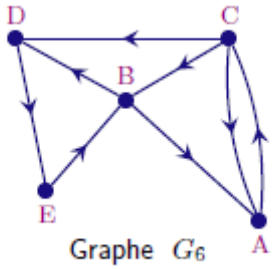
Exercice n° 3.

On considère la classe `Graphe` (avec dictionnaire d'adjacence).

1. Ajouter une méthode `afficher`, pour afficher le graphe sous la forme suivante :
   0 [1, 3]
   1 [2, 3]
   2 [1]
   3 [3]
2. Ajouter une méthode `nb_sommet()`, qui renvoie le nombre de sommets du graphe.
3. Ajouter une méthode `degre(s)`, qui renvoie le degré d'un sommet `s` du graphe.
   En déduire une méthode `nb_arete()`, qui renvoie le nombre total d'arêtes du graphe.
4. Ajouter une méthode `supprimer_arete(s1, s2)`, pour supprimer l'arête entre les sommets `s1` et `s2` du graphe. S'il n'y a pas d'arête entre ces deux sommets, la méthode n'a aucun effet.

Exercice n° 4.

On considère le graphe orienté G_6 :



Graphe G_6

1. Dérouler à la main le parcours en profondeur sur le graphe à partir des sommets A, B et C. Donner à chaque fois la valeur finale de la liste `vus`
2. Dérouler à la main le parcours en largeur sur le graphe à partir des sommets A, B et C. Donner à chaque fois la valeur finale de la liste `dist`

Exercice n° 5.

On peut se servir d'un parcours en profondeur pour détecter si un graphe non orienté est connexe, c'est à dire si tous ses sommets sont reliés entre eux par un chemin.
Pour cela, il suffit de faire un parcours en profondeur à partir d'un sommet quelconque, puis de vérifier que lors de ce parcours tous les sommets ont été atteints.
Écrire une fonction `est_connexe()` qui réalise cet algorithme.

Exercice n° 6.

Nous avons vu dans le cours une fonction qui permet de déterminer, à partir d'un parcours en profondeur, s'il existe un chemin entre deux sommets `u` et `v` d'un graphe. Nous allons ici déterminer un tel chemin s'il existe, toujours à partir d'un parcours en profondeur. Pour cela on codera deux fonctions :

1. La première est très proche de la fonction vue en cours `parcours_profondeur` :

```
def parcours_profondeur_chemin(g, org, s, vus = {}):
    """parcours depuis le sommet s en venant su sommet org"""

```

   Mais ici `vus` sera un dictionnaire dont les clés seront les sommets visités et les valeurs associées : le sommet qui a permis de l'atteindre pendant le parcours.

   Cette fonction prend en paramètre un argument supplémentaire : `org`, qui est le sommet dont on vient en parcourant l'arête : `org` $\rightarrow$ `s`

2. La fonction `chemin` renvoie un chemin de `u` à `v`, s'il existe, `None` sinon.

   Pour cela lancer la fonction `parcours_profondeur_chemin` à partir du sommet `u`, en donnant au paramètre `org` la valeur `None`, puis si le sommet `v` a été atteint, construire le chemin dans une liste en remontant le dictionnaire `vus` de `v` jusqu'à `u`.

```
def chemin(g, u, v):
    """un chemin de u à v, s'il existe, None sinon"""

```

Exercice n° 7.

Utilisons dans cet exercice le parcours en largeur pour déterminer, s'il existe, un chemin de longueur minimale entre deux sommets.
Pour cela on codera deux fonctions :

1. La première est très proche de la fonction vue en cours `parcours_largeur` :

```
def parcours_lareur_chemin(g, source):
    """parcours depuis le sommet source"""

```

   Mais ici le dictionnaire `vus` remplacera le dictionnaire `dist`. Dans `vus` les clés seront les sommets visités et les valeurs associées : le sommet qui a permis de l'atteindre pendant le parcours. Au sommet `source` on associe la valeur `None`

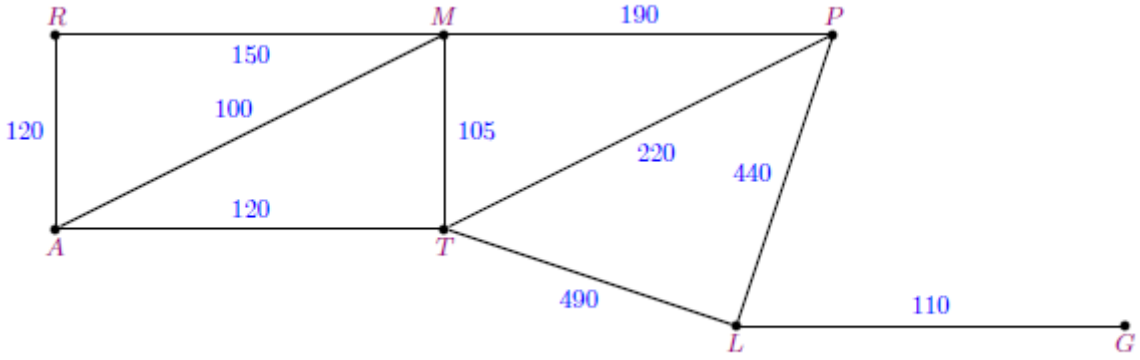
2. La fonction `chemin` renvoie un chemin de `u` à `v` réalisant la distance, s'il existe, `None` sinon.

   Pour cela lancer la fonction `parcours_largeur_chemin` à partir du sommet `u`, si le sommet `v` a été atteint, construire le chemin dans une liste en remontant le dictionnaire `vus` de `v` jusqu'à `u`.

```
def chemin(g, u, v):
    """un chemin de u à v, s'il existe, None sinon"""

```

Exercice n° 8.

La recherche de chemin dans un graphe est quelque chose de très utile : vos logiciels de guidage par GPS utilisent de tels algorithmes pour vous préparer un itinéraire optimisé en distance ou en temps pour aller d'une ville A à une ville B.
On représente le réseau autoroutier entre les villes de Rennes (R), Angers (A), Tours (T), Le Mans (M), Paris (P), Lyon (L) et Grenoble (G) à l'aide d'un graphe. Les villes sont les sommets du graphe et les (auto)routes sont représentées par les arêtes du graphe. On a indiqué les distances entre les villes sur les arêtes.



1. On définit pour matrice d'adjacence de ce graphe, la matrice telle que le terme a_{ij} est le nombre de kilomètres séparant les villes numéro i et numéro j.
   Écrire cette matrice d'adjacence
2. Écrire une fonction `matrice2liste(matrice, noms)`
   - prenant en paramètres `matrice` (La matrice d'adjacence du graphe) et `noms` (la liste des noms des sommets dans l'ordre de la matrice)
   - renvoyant un dictionnaire dont les clés sont les sommets et les valeurs sont un tableau de tuples au format `('Nom', distance)`.

   Exemple : `A` est reliée à `M, R et T`. Le dictionnaire commencera par {`'A':[('M',100), ('R', 120), ('T', 120)] ...`}

```
def matrice2liste(matrice, noms):
    """Renvoie la liste d'adjacence sous format dictionnaire"""

    # YOUR CODE HERE
    raise NotImplementedError()
matrice = [[1, 2, 1, 0],
           [2, 3, 0, 1],
           [1, 0, 0, 0],
           [0, 1, 0, 4]]
test = matrice2liste(matrice, ['A', 'B', 'C', 'D'])
assert test['C'] == [('A', 1)]
assert ('B', 2) in test['A']
assert ('D', 4) in test['D']
```

3. On considère à présent le graphe représentant les mêmes villes mais les sommets sont pondérés par le temps de parcours en minutes. Voici sa liste d'adjacence au format dictionnaire tel que décrit plus haut :

```
{'A': [('M', 65), ('R', 90), ('T', 80)],
 'G': [('L', 70)],
 'L': [('G', 70), ('P', 230), ('T', 260)],
 'M': [('A', 65), ('P', 95), ('R', 90), ('T', 55)],
 'P': [('L', 230), ('M', 95), ('T', 130)],
 'R': [('A', 90), ('M', 90)],
 'T': [('A', 80), ('L', 260), ('M', 55), ('P', 130)]}
```

   Écrire une fonction `liste2matrice(dico)` prenant en paramètre un graphe donné par une liste d'adjacence sous format dictionnaire comme ci-dessus et renvoyant la matrice d'adjacence de ce graphe ainsi que un tableau des sommets.

   En d'autre termes, la fonction `liste2matrice` est l'inverse de la fonction `matrice2liste` précédente.

```
def liste2matrice(dico):
    """Converti une liste d'adjacence en matrice d'adjacence"""
    # YOUR CODE HERE
    raise NotImplementedError()
# Vérification
test = {'A': [('A', 1), ('B', 2), ('C', 1)],
 'B': [('A', 2), ('B', 3), ('D', 1)],
 'C': [('A', 1)],
 'D': [('B', 1), ('D', 4)]}
l, n = liste2matrice(test)
assert n == ['A', 'B', 'C', 'D']
assert l == [[1, 2, 1, 0], [2, 3, 0, 1], [1, 0, 0, 0], [0, 1, 0, 4]]
```