Chapitre (ALGO) 8 Graphes

Rien ne se passe dans l'Univers sans qu'un minimum ou un maximum apparaisse.

— Leonhard Euler

Résumé & Plan

Les graphes constituent un outil de modélisation très utilisé en Informatique (description du réseau internet, relations sur les réseaux sociaux, ...) et en Mathématiques (graphes probabilistes, ...). L'objectif est d'étudier l'objet, puis de l'implémenter en Python, et enfin savoir les parcourir.

- Les chapitres d'Informatique sont composés de cours et d'exercices intégrés. Le cours sera projeté au tableau.
- Il n'est pas attendu que toute la classe aborde tous les exercices. Traitez donc en priorité les exercices présents dans la liste donnée à chaque début de séance.
- Exercices 💣 / **Pour aller plus loin** : exercices plus difficiles, ou plus techniques. À ne regarder que si les autres sont bien compris.

Fichier externe ?

OUI `TP_Graphes.py` *(présents dans le répertoire partagé de la classe)*

Exercice 1 | **Fichier externe** Commencez par récupérer sur le réseau du lycée le fichier `TP_Graphes.py` qui contient certains bouts de codes nécessaires à ce TP. Le placer dans votre répertoire personnel et enregistrez-le : vous travaillerez directement ce fichier python.

1. GÉNÉRALITÉS

De nombreux problèmes algorithmiques se ramènent à l'étude de graphes par l'intermédiaire d'une modélisation adaptée (résolution d'un jeux logique du type sudoku ou rubik's cube, développement d'une « intelligence artificielle » pour jouer aux échecs, ...). Les problèmes intéressants correspondent à des graphes gigantesques dont le traitement ne peut être effectué correctement et en un temps raisonnable qu'à l'aide d'algorithmes optimisés.

1.1. Introduction & Motivation

Dans ce chapitre, nous nous intéressons à un nouveau type de structures de données, les « graphes ». Cette structure est utilisée pour représenter des *relations* dans un *ensemble d'objets homogènes* :

- représentation de réseaux (réseaux routiers, de machines, sociaux, *etc.*)
- représentation de l'ordre d'exécution de tâches (tâches physiques, réalisation de processus sur une de machine, *etc.*)

Les graphes sont également utilisées dans les algorithmes d'optimisation :

- algorithmes d'ordonnancement de tâches (trouver l'ordre d'exécution optimal),
- algorithmes de routage réseau (calcul de chemin,...)
- algorithmes d'intelligence artificielle pour les jeux (Min-Max, ...).

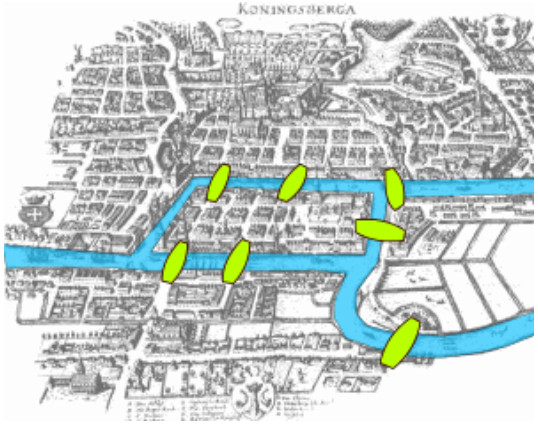
1.2. Utilisation des graphes

Les situations nécessitant l'utilisation de graphes sont très nombreuses et couvrent tous les domaines.

- L'intérêt des graphes est de pouvoir représenter différentes situations avec une même structure.

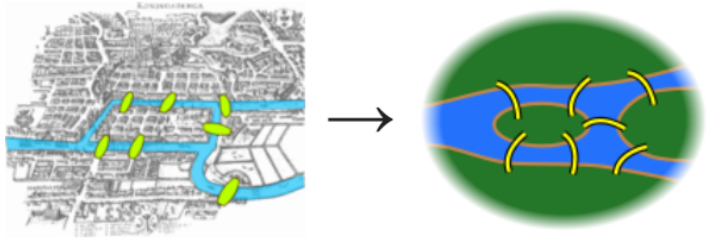
- Certains problèmes liés à ces situations diverses vont donc pouvoir se traduire comme des questions à résoudre sur les graphes.
- En appliquant des outils mathématiques et informatiques à ces graphes, nous pourrons résoudre les problèmes concrets liés à ces situations.

ORIGINES : LE PROBLÈME DES 7 PONTS. L'origine de la théorie des graphes est liée à un problème formulé par le grand mathématicien suisse Leonhard EULER connu sous le nom du « problème des sept ponts de Königsberg ». Le problème consiste à trouver une promenade à partir d'un point donné qui fasse revenir à ce point en passant une fois et une seule par chacun des sept ponts de la ville de Königsberg (voir ci-dessous).



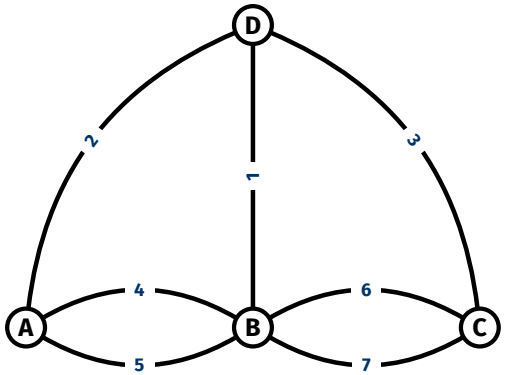
LES SEPT PONTS DE KONIGSBERG

Dans un article publié en 1741, EULER jette les bases de la théorie des graphes et trouve une solution mathématique au problème en appliquant des résultats de sa théorie à la situation étudiée et prouve que le problème n'a pas de solution.



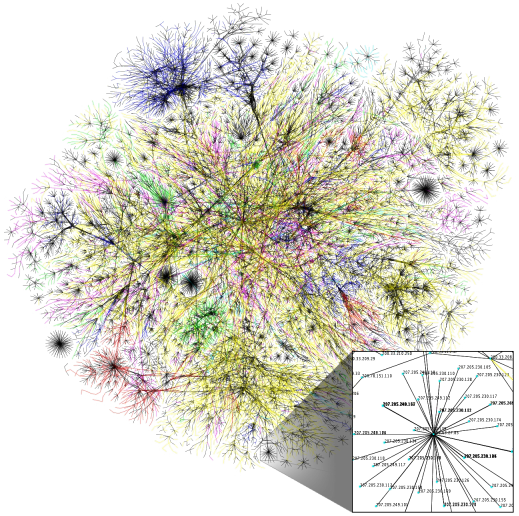
MODÉLISATION DES SEPT PONTS DE KONIGSBERG

On arrive tout droit au graphe ci-après ; les arêtes désignant les ponts, et les sommets étant des lieux situés entre les ponts.



MODÉLISATION DES SEPT PONTS SOUS FORME DE GRAPHE

Le réseau Internet est modélisable en un graphe dans lequel existent des relations (ici les connexions) dans l'ensemble des machines connectées.



GRAPHE PARTIEL DE L'INTERNET, BASÉ SUR LES DONNÉES DU PROJET OPTE DU 15 JANVIER 2005

1.3. Vocabulaire

Commençons par introduire le vocabulaire nécessaire sur les graphes.

Définition 1 | Graphe non pondéré

- On appelle *graphe* la donnée d'un ensemble S de points appelés *sommets* et d'un ensemble de lignes (*resp.* flèches) $\mathscr{A}$ appelées *arêtes* (*resp. arcs* ou encore *arêtes orientées*) qui relient certains sommets entre eux (une partie de $S \times S$ en pratique) : $\mathscr{G} = (S, \mathscr{A})$.
- Le nombre de sommets d'un graphe s'appelle l'*ordre* du graphe.
- ◇ Deux sommets reliés entre eux par une arête (non orientée, ou deux orientées) sont dits *adjacents*. Autrement dit, $s_1 \in S$ et $s_2 \in S$ sont adjacents si :
  $$(s_1, s_2) \in \mathscr{A} \text{ et } (s_2, s_1) \in \mathscr{A}.$$
  - ◇ Un sommet s_2 est dit *voisin de* s_1 s'il existe une arête menant de s_1 à s_2, c'est-à-dire si : $(s_1, s_2) \in \mathscr{A}$.
- ◇ On dit qu'un graphe est *non-orienté* si chaque arête peut être parcourue dans les deux sens. Autrement dit,
  $$\forall (s_1, s_2) \in \mathscr{A}, \quad (s_2, s_1) \in \mathscr{A}.$$
  - ◇ On dit qu'un graphe est *orienté* s'il existe au moins une arête qui peut être parcourue que dans un sens, c'est-à-dire il existe un couple de sommets $(s_1, s_2) \in S^2$ tel que :
  $$(s_1, s_2) \in \mathscr{A} \text{ et } (s_2, s_1) \notin \mathscr{A}.$$

  Note : *En général, on représente alors un tel graphe avec que des flèches au lieu d'un simple trait, voir la prochaine remarque pour plus de détails.*
- Dans un graphe non-orienté, le *degré* d'un sommet est le nombre d'arêtes issues de ce sommet, avec pour convention les boucles comptées deux fois.
- Un sommet qui n'est rattaché à aucune arête (sortante, entrante ou simple) est dit *isolé*.
- Une *boucle* est une arête reliant un sommet à lui-même, c'est-à-dire un élément de $\mathscr{A}$ de la forme (s, s) avec $s \in \mathscr{A}$.
- Une *arête multiple* est une arête reliant plusieurs fois un même couple de sommets, c'est-à-dire un couple $(s_1, s_2) \in S^2$ apparaissant au moins deux fois dans $\mathscr{A}$.
- Un graphe est dit *simple* s'il ne contient ni boucles, ni arêtes multiples.

Définition 2 | Graphe pondéré

Un graphe $\mathscr{G} = (S, \mathscr{A})$ est dit *pondéré* si chaque arête est associée à un nombre réel positif appelé *poids*, c'est-à-dire si dans la définition précédente on remplace « $\mathscr{A}$ est une partie de $S \times S$ » par « $\mathscr{A}$ est une partie de $S \times S \times \mathbb{R}^+$.

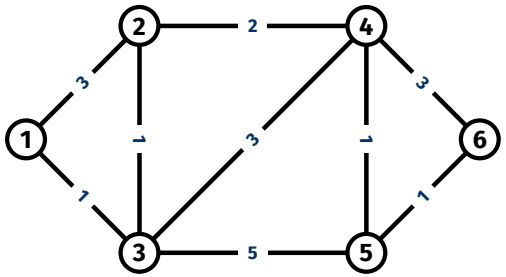
On pourrait aussi considérer des poids négatifs, mais ce ne sera pas le cas dans ce cours.

Remarque 1 (Représentation graphique) Dans la suite, nous n'écrirons que très rarement la description ensembliste des graphes (ensemble de type « arêtes/arcs » × « couples de sommets »). On se contentera de les définir à l'aide d'un dessin qui comporte généralement :

- des cercles possédant un nom (une « étiquette »), correspondant aux sommets donc aux éléments de S,
- des traits (avec ou sans flèches, avec ou sans poids) reliant des couples de sommets et correspondant aux arêtes donc aux éléments de $\mathscr{A}$. Si $(i, j) \in \mathscr{A}$ et $(j, i) \in \mathscr{A}$ (avec $i \neq j$) on reliera les deux sommets avec un trait, si seulement l'un des deux couples est dans $\mathscr{A}$, on les reliera par une flèche.

Note : *Une remarque importante s'impose d'emblée : la position géométrique des sommets dans le plan, ainsi que le positionnement des arêtes n'ont aucune importance pour ce qui nous intéresse dans la suite.*

Exemple 1 On considère le graphe pondéré suivant donné par sa représentation graphique.



Le graphe $\mathscr{G} = (S, \mathscr{A})$ est ici défini par :

- l'ensemble des sommets $S = [\![1, 6]\!]$,
- l'ensemble des arêtes $\mathscr{A} = \{(3,4,3), (4,3,3)\ldots, (3,5,5), (5,3,5)\ldots\}$.

Le graphe présenté ici possède les caractéristiques suivantes :

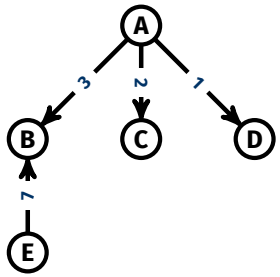
- Il n'est pas *orienté*.
- Il est *simple* : il y a au plus une arête entre deux sommets et aucune boucle.
- Il est *pondéré* : à chaque arête on associe une valeur positive appelée *poids* ou *distance entre les sommets*. Valeur indiquée sur l'arête.

Exercice 2 |

1. Représenter le graphe associé aux ensembles ci-dessous.

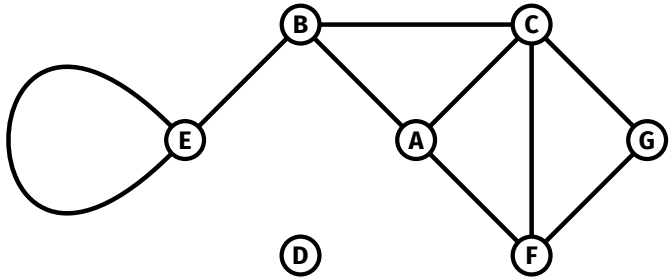
$$S = [\![0, 4]\!],$$
$$\mathscr{A} = \{(0,1), (1,0), (1,4), (4,1), (0,2), (2,0), (0,3), (3,0)\}.$$

2. Donner les ensembles S et $\mathscr{A}$ associés au graphe ci-dessous.



Exercice 3 |

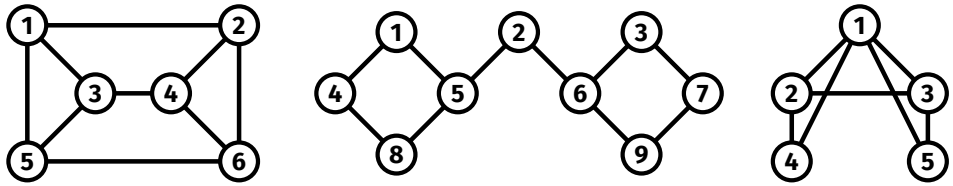
On considère le graphe ci-après.



1. Citez des couples de sommets voisins et d'autres qui le ne sont pas.
2. Donnez les degrés des sommets A, B, D, E.
3. Citez un sommet qui comporte une boucle.

Exercice 4 | Somme des degrés dans un graphe

On considère les trois graphes ci-dessous, de manière générale un graphe non orienté.



1. Pour chacun de ces trois graphes, calculer la somme des degrés de tous les sommets, ainsi que le nombre d'arêtes. Que peut-on conjecturer ?
2. Démontrer cette conjecture.
3. Que peut-on en déduire concernant la parité de la somme des degrés de tous les sommets d'un graphe ?

Exemple 2 (Réseaux sociaux)

- Prenons l'exemple des réseaux sociaux tels sur **Facebook** ou son alternative libre **Diaspora**. Dans ce type de réseau social :
  - ◇ les sommets sont les utilisateurs,
  - ◇ les arêtes sont les liens d'amitié entre ces utilisateurs.

  Est-il orienté ? L'amitié se veut être un lien bidirectionnel, on ne peut être ami de quelqu'un qui n'est pas votre ami : c'est un graphe non-orienté.
- Au contraire dans les réseaux de **microblogging** comme **Twiter** ou son alternative libre **Mastodon**, les liens ne sont pas de la même nature. On peut suivre

une personne, mais il n'est pas obligatoire que cette personne vous suive en retour.

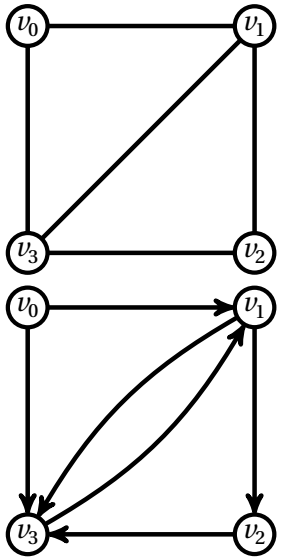
1.4. Chemin et cycle

Définition 3 | Chemin sur un graphe & Connexité

Soit $\mathcal{G} = (S, \mathcal{A})$ un graphe pondéré ou non, et u, v deux sommets.

- On appelle *chemin reliant un sommet u à un sommet v* toute suite finie de sommets reliés deux à deux par des arêtes et menant de u à v.
- On appelle *poids de c* la somme des coefficients des arêtes lorsque le graphe est pondéré, et le nombre d'arêtes sinon. La *longueur* est le nombre d'arêtes intervenant dans le chemin, c'est-à-dire le nombre de sommets moins un.
- Un chemin est dit *simple* s'il n'emprunte pas deux fois la même arête. Un chemin simple reliant un sommet à lui-même et contenant au moins une arête est appelé un *cycle*.

Exemple 3 Explicitons quelques chemins sur les deux graphes ci-après.

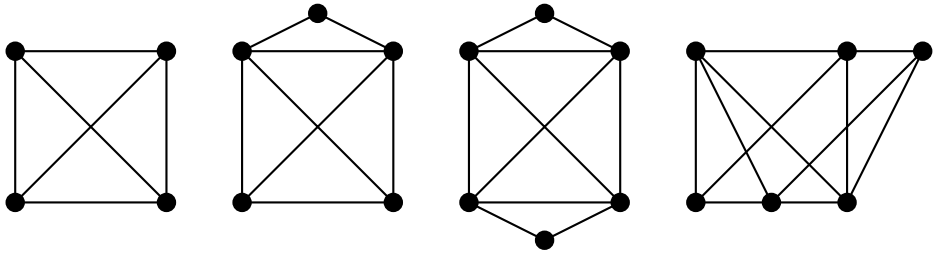


$v_0 \to v_1 \to v_2 \to v_3$, $v_0 \to v_1 \to v_3$ et $v_0 \to v_3$ sont des chemins simples du sommet v_0 vers le sommet v_3 de longueurs respectives 3,2 et 1. En revanche, $v_0 \to v_1 \to v_0$ n'est pas un chemin simple ; ce n'est pas un cycle.

$v_0 \to v_1 \to v_2$ et $v_0 \to v_3 \to v_1 \to v_2$ sont des chemins du sommet v_0 vers le sommet v_2 de longueurs respectives 2 et 3. En revanche, il n'existe pas de chemin de v_2 vers v_0. Le chemin $v_1 \to v_2 \to v_3 \to v_1$ est un cycle.

Exercice 5 | Théorème d'Euler

1. Commençons par un petit jeu ! Est-il possible de tracer les figures suivantes sans lever le crayon et sans passer deux fois sur le même trait ?



2. Dans un graphe connexe (c'est-à-dire en une seule partie), un chemin qui parcourt toutes les arêtes une et une seule fois est appelé chemin eulérien. On peut démontrer qu'un « graphe connexe possède un chemin eulérien si et seulement si tous les sommets sont de degré pair, sauf au plus 2 ». Réfléchir à nouveau à la question 1 en utilisant le théorème d'Euler.

Cadre

Dans la suite, on supposera toujours $\mathcal{G}$ d'ordre fini, et le plus souvent (en 1ère année) nous travaillerons avec des graphes simples non pondérés, orientés ou non.

2. IMPLÉMENTATION INFORMATIQUE

Venons-en à présent aux différentes manières de « coder » un graphe en Python. Nous en présentons deux, on peut :

- **[Représentation par dictionnaire de voisins]** réer un dictionnaire de clefs les noms des sommets, et valeurs la liste des voisins associés. Si le graphe est pondéré, on peut indiquer une liste de couples en valeurs (la seconde coordonnée étant le poids du voisin associé).
- **[Représentation par matrice d'adjacence]** On peut aussi indiquer dans une matrice si un sommet est voisin d'un autre ou pas ; plus précisément indiquer en position (i, j) un 1 s'il existe une arête $i \to j$. *(Il est bien entendu aussi possible d'y indiquer des booléens `True`, `False` si l'on préfère)*

Nous commençons par utiliser la première représentation.

2.1. Dictionnaire des voisins

Dans cette partie, nous allons stocker le graphe en utilisant un dictionnaire ; c'est cette représentation qui est la plus utilisée pour diverses raisons pratiques ; tant au

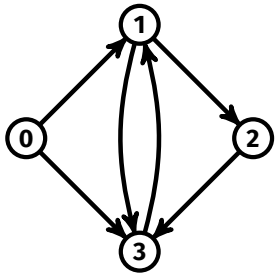
niveau de la facilité d'écriture qu'au niveau de la rapidité des algorithmes de parcours de graphes que nous allons voir plus tard.

Définition 4 | Dictionnaires des voisins/poids

Soit $\mathcal{G} = (S, \mathcal{A})$ un graphe d'ordre $n \in \mathbb{N}^*$. On appelle *dictionnaire des voisins associé à $\mathcal{G}$*, le dictionnaire de clés les sommets du graphe, et de valeur associée la liste de ses sommets voisins (avec pour convention la liste vide s'il n'y en a pas).

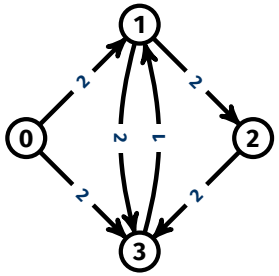
Exemple 4

- Voici un premier exemple de graphe, et son dictionnaire associé.



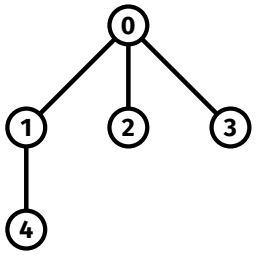

```
D = {0:[1,3], 1:[2,3], 2:[3],
3:[1]}
```

- Considérons maintenant plutôt un graphe pondéré. Dans ce cas, plutôt que d'introduire une liste de sommets en valeur du dictionnaire, on indiquera à la place une liste de couples contenant le numéro de sommet voisin, ainsi que son poids. Par exemple,

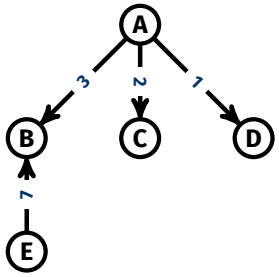



```
D = {0 : [(1, 2), (3, 2)], 1 :
[(2, 2), (3, 2))], 2 : [(3, 2)],
3 : [(1, 1)]}
```

Exercice 6 | Implémenter, directement dans votre éditeur Pyzo et sous forme de dictionnaire des voisins, les graphes ci-dessous.



GRAPHE G_1

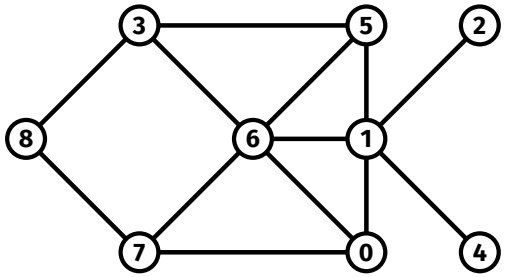


GRAPHE G_2

À partir de plusieurs instructions à taper directement dans la console : récupérer les voisins de 2 pour `G_1`, et les voisins de `A` pour `G_2` (pour la seconde, on pourra utiliser une liste par compréhension).

Remarque 2 (Cas de sommets numérotés à partir de zéro) Notons que, dans le cas où les sommets sont des entiers successifs à partir de 0, on peut choisir de stocker ces informations dans une liste au lieu d'un dictionnaire. La *liste d'adjacence* est alors une liste de listes, l'élément d'indice i contenant là encore la liste des sommets voisins du sommet i. Par exemple, le graphe précédent aurait été stocké par la liste d'adjacence suivante : `[[1,3], [2,3], [3], [1]]`.

Exercice 7 | Quelques fonctions utilisant cette implémentation Dans le fichier `TP_Graphes.py`, vous avez le dictionnaire codant ce graphe :



GRAPHE G_3

Pour simplifier, on suppose ici que `G` désigne un graphe ***non pondéré****.*

1. Écrire une fonction `sommets(G)` qui prend en argument un dictionnaire codant un graphe, et renvoyant la liste des sommets.
2. Écrire une fonction `voisins(G, s)` qui prend en argument un dictionnaire codant un graphe et un sommet `s`, et renvoyant la liste des voisins de `s` si `s` est un sommet du graphe, et `None` dans le cas contraire.

3. Écrire une fonction `nb_arcs(G)` qui prend en argument un dictionnaire codant un graphe et renvoyant le nombre d'arcs (c'est-à-dire le nombre d'arêtes orientées) présentes dans le graphe. En supposant que le graphe n'est pas orienté, comment obtenir le nombre d'arêtes ?
4. Écrire une fonction `sans_vois(G)` qui renvoie la liste des sommets n'ayant aucun voisin.
5. 💣 Écrire une fonction `est_oriente(G)` qui renvoie `True` si le graphe est orienté, et `False` dans le cas contraire.
6. Testez vos fonctions sur le graphe G_3.

2.2. Matrice d'adjacence

Définition 5 | Matrice d'adjacence

Soit $\mathcal{G} = (S, \mathcal{A})$ un graphe d'ordre $n \in \mathbb{N}^\star$ dont les sommets sont numérotés entre 0 et $n-1$. On appelle *matrice d'adjacence* de $\mathcal{G}$ la matrice $A = (a_{i,j})_{(i,j)\in S^2} \in \mathcal{M}_n(\mathbb{R})$ telle que pour tout couple de sommets $(i,j) \in S^2$:

- si $(i,j) \in \mathcal{A}$, c'est-à-dire s'il existe une arête $i \to j$, alors $a_{i,j} = 1$,
- si $(i,j) \notin \mathcal{A}$ alors $a_{i,j} = 0$.

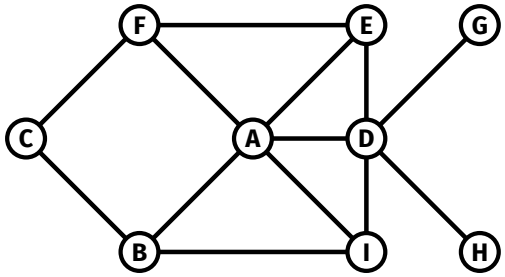
Exemple 5 On reconsidère le graphe G_3 précédent. Les arêtes peuvent être stockées dans une matrice : on y indique des 1 (ou les poids, dans le cas pondéré) aux coefficients correspondant à des sommets reliés, et des zéro ailleurs. On obtient ici : $A = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \end{array} \begin{array}{c} \begin{array}{ccccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} \end{array}$.

On constate que c'est une matrice symétrique, cela provient de la non-orientation du graphe (chaque arête (u,v) fournit une arête (v,u) de même poids). On peut coder cette matrice à l'aide du module `numpy`, comme nous l'avions fait dans le Chapitre (NUM) 3.

```
import numpy
G_3m = np.array([[0,  1,  0,  0,  0,  0,  1,  1,  0],
    [1,  0,  1,  0,  1,  1,  1,  0,  0],
    [0,  1,  0,  0,  0,  0,  0,  0,  0],
    [0,  0,  0,  0,  0,  1,  1,  0,  1],
    [0,  1,  0,  0,  0,  0,  0,  0,  0],
    [0,  1,  0,  1,  0,  0,  1,  0,  0],
    [1,  1,  0,  1,  0,  1,  0,  1,  0],
    [1,  0,  0,  0,  0,  0,  1,  0,  1],
    [0,  0,  0,  1,  0,  0,  0,  1,  0]])
```

Exercice 8 | Quelques fonctions utilisant cette implémentation Dans le fichier `TP_Graphes.py`, vous avez la matrice précédente codant le graphe G_3.
Reprendre toutes les fonctions précédentes sauf `est_oriente` (appelées `sommets_m`, `voisins_m` *etc.*) mais où cette fois-ci `G` est supposé être une matrice d'adjacence. Testez vos fonctions sur `G_3m`. *On pourra se servir de `np.sum` pour la dernière fonction. Pour simplifier, on supposera à nouveau que `G` désigne un graphe non pondéré.*

NUMÉROTATION DES SOMMETS & DICTIONNAIRE D'ÉTIQUETAGE. Nous avons supposé les sommets numérotés de manière bien particulière (à partir de zéro). Si l'on souhaite utiliser des lettres ou tout autre chose, on fera appel à un *dictionnaire d'étiquetage* qui donnera une correspondance entre les entiers et les vrais noms de sommets. Par exemple, si l'on s'intéresse à :



On peut alors par exemple numéroter ainsi (pour que ça corresponde aux anciens sommets) :

```
>>> D = {'A':6,'B':7,'C':8,'D':1,'E':5,'F':3,'G':2, 'H':4,'I':0}
```

Ce dictionnaire permet alors d'accéder rapidement à l'entier correspondant à chaque sommet, par exemple pour `'A'` :

```
>>> D['A']
6
```

Si l'on souhaite savoir si S_1 est voisin de S_2, on analysera le résultat de :

```
G[D[S_1], D[S_2]]
```

Remarque 3 De manière équivalente à ce dictionnaire `D`, on pourrait se donner une liste `sommets` qui contient la liste des sommets numérotés dans le bon ordre : `sommets = ['I', 'D', 'G', 'F', 'H', 'E', 'A', 'B', 'C']`.

En résumé,

représentation par MATRICE D'ADJACENCE	=	une matrice de taille le nombre de sommets indiquant la présence d'arêtes	+ (un dictionnaire d'étiquetage, si nécessaire)
représentation par DICTIONNAIRE DES VOISINS	=	un dictionnaire de	{ CLEFS : les sommets $s \in S$; VALEURS : les voisins de s

Cadre

Dans toute la fin du cours, on utilisera une description des graphes en terme de dictionnaire des voisins. Les fonctions s'adaptent facilement aux matrices d'adjacence.

2.3. Coloriage de graphes & Application

Voici deux exemples classiques d'utilisation des graphes.

Exercice 9 | Graphe pour représenter des problèmes d'incompatibilité Dans une classe de première, chaque élève a choisi ses spécialités : Maths (M), Humanité (H), NSI (N), Physique-Chimie (P), SVT (S) et Langues (L).

- Certains suivent M, H, N, d'autres N, P, S et d'autres S, L, P. Tous les examens ont une durée de 2h et ont lieu la même demi-journée.
- La question que l'on se pose est la suivante : comment établir un horaire de telle manière que la durée totale d'épreuves soit la plus courte possible ?
- On peut modéliser la situation à l'aide d'un graphe. Dans cette modélisation les sommets sont les spécialités : M, H, N, P, S, L.
- On relie deux sommets par une arête lorsqu'il existe un élève ayant choisit les deux sommets en spécialité. Par exemple, M et H sont reliés car certains élèves suivent à la fois M et H.

1. Dessiner le graphe correspondant.

2. On cherche à attribuer à chaque cours un créneau horaire de 2h. La solution consiste à colorier les sommets adjacents avec des couleurs différentes en utilisant un minimum de couleurs, une couleur représentant alors un créneau donné de 2h. Combien de créneaux doit-on prévoir au maximum pour organiser les cours ?

Exercice 10 | Algorithme de coloration On considère ici un graphe non orienté, pour lequel on veut associer à chacun de ses sommets une couleur de telle sorte que deux sommets voisins aient une couleur différente, et en tâchant d'utiliser le moins de couleurs possibles.

Un célèbre théorème, dit des *quatre couleurs*, affirme qu'il est toujours possible d'obtenir un tel coloriage en utilisant quatre couleurs au plus, mais il n'existe pas d'algorithme efficace permettant d'obtenir un tel coloriage optimal à tous les coups.

Voici néanmoins un algorithme conduisant généralement à une bonne solution, même si celle-ci n'est pas toujours optimale (au sens du nombre de couleurs utilisées). Il repose sur un principe d'algorithme « glouton » de la manière suivante :

- les couleurs sont représentées par des entiers successifs à partir de 0 ;
- on décrit un à un tous les sommets (dans l'ordre que l'on souhaite), et on associe au sommet courant la plus petite couleur non déjà utilisée pour ses sommets voisins.

1. Écrire une fonction `premierNonPresent(L)` renvoyant, pour une liste `L` formée d'entiers naturels, le plus petit entier positif non présent dans `L`. Par exemple, `premierNonPresent([0, 3, 2])` renverra `1`, et `premierNonPresent([0, 1, 2])` renverra `3`.

2. On dispose alors des fonctions `premierNonPresent` et `voisins`. Compléter le code suivant de la fonction `coloriage(G,S)`, où `G` désigne un graphe (sous forme de dictionnaire de voisins) et `S` désigne une liste contenant ses sommets dans un certain ordre, et renvoyant un dictionnaire associant à chaque sommet sa couleur.

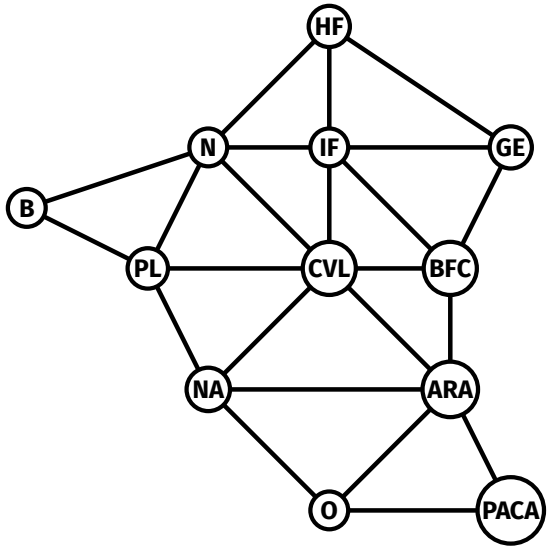
```
def coloriage(G, S) :
    """
    renvoie le dictionnaire correspondant au coloriage des \
    ↪ sommets du graphe G pour le sens de parcours précisé dans S
    """
    C = {}
    # On parcourt les sommets dans l'ordre de S.
    for s ________ :
    # Construction de la liste U des couleurs déjà utilisées par \
    ↪ les voisins de s.
        U = []
        for vois in __________ : # pour chaque voisin de s
                if vois in ______ : # s'il a déjà une couleur
                    U.append(________) # on ajoute cette couleur \
                    ↪ à la liste

        # Recherche de la plus petite couleur non utilisée par \
        ↪ les voisins et association de cette couleur à s dans \
        ↪ le dictionnaire
        C[s] = _______________
    return _______________
```

3. On considère le graphe des régions de France métropolitaine continentale adjacentes :

Tester l'algorithme précédent sur les ordres de parcours `S1` et `S2` donnés par :

```
S1 = ['HF', 'N', 'IF', 'GE', 'B','PL','CVL','BFC','NA','ARA','O','PACA']
S2 = ['ARA','BFC','B','CVL','GE','HF','IF','N','NA','O','PL','PACA']
```

Que peut-on constater ?



CARTE DE FRANCE MODÉLISÉE PAR UN GRAPHE DE SOMMETS LES RÉGIONS

3. PARCOURS EN LARGEUR

Dans les sections précédentes, nous avons étudié les graphes et nous avons vu quelques implémentation possibles en Python, notamment par dictionnaire des voisins que nous allons largement utiliser ici. Un des premiers algorithmes qu'on doit savoir utiliser sur un graphe est celui de son parcours (un analogue de boucle `for` en quelque sorte, mais adaptée aux graphes), c'est ce que nous voyons à présent.

3.1. Objectif

Parcourir un graphe, c'est visiter ses différents sommets, en partant d'un sommet quelconque, afin de pouvoir opérer une action tour à tour sur eux.

Ces algorithmes de parcours d'un graphe sont à la base de nombreux algorithmes très utilisés : routage des paquets de données dans un réseau, découverte du chemin le plus court pour aller d'une ville à une autre, trouver la sortie d'un labyrinthe...

Le choix d'un parcours dépend essentiellement du contexte et des attentes d'un problème. Cette description sommaire des parcours révèle une autre différence essentielle entre les graphes et les structures de données linéaires (comme les listes). En raison de l'existence de cycles, un parcours peut ramener à un sommet déjà visité.

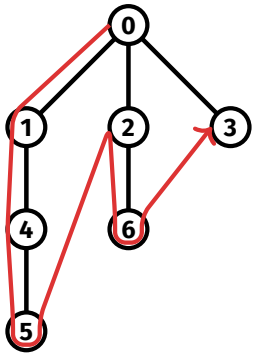
Pour en tenir compte et ne pas recommencer un parcours déjà fait, il convient de garder une information sur le fait qu'un sommet ait été visité ou pas lors d'un parcours. Cette information peut être stockée dans une structure de données adaptée.

Cadre

Dans la suite, nous supposons le graphe simple *(sans arête multiple ou boucle)* **non orienté et connexe.** *(pour deux sommets quelconques il existe toujours un chemin dans le graphe qui les relie)*

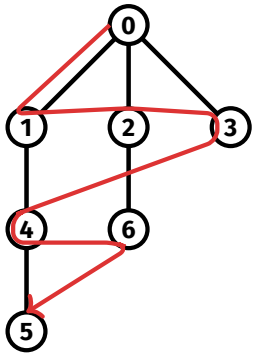
Remarque 4 **(Différence entre parcours en largeur & profondeur)** Il existe deux manières principales de parcourir ainsi les sommets d'un graphe non pondéré :

- le parcours en profondeur, que l'on utilise par exemple pour explorer un labyrinthe. Ce parcours sera vu en 2ème année, l'objectif est d'avancer sur le graphe tant qu'on est pas bloqué, puis de rebrousser chemin en cas de blocage.
- Le parcours en largeur, que l'on étudie dans cette section et dont l'objectif est d'explorer les sommets par distance croissante depuis l'origine.
- En 2ème année, vous verrez une généralisation du parcours en largeur aux graphes pondérés (algorithmes de DIJKSTRA).



Ordre souhaité : $(0,1,4,5,2,6,3)$

PARCOURS EN PROFONDEUR



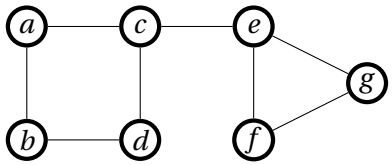
Ordre souhaité : $(0,1,2,3,4,6,5)$

PARCOURS EN LARGEUR

DEUX TYPES DE PARCOURS

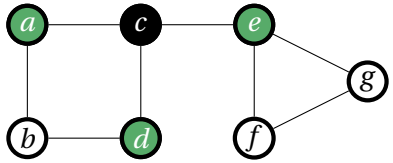
EXEMPLE DE PARCOURS EN LARGEUR D'UN GRAPHE NON-ORIENTÉ. Le parcours en largeur d'un graphe à partir d'un sommet donné consiste à commencer par explorer les voisins du sommet, puis les voisins de ses voisins, puis les voisins des voisins de ses voisins, et ainsi de suite.

Considérons le graphe ci-dessous. Déroulons cet algorithme de parcours par exemple à partir du sommet c.

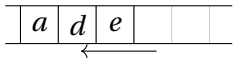


Exercice 11 | À partir du sommet `c`, quelle est la liste souhaitée en sortie d'algorithme ?

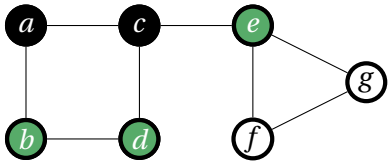
On marque c comme *visité*, puis ses sommets adjacents sont découverts, marqués en gris clair et mémorisés dans ce que l'on appelle une *file* (indiquée sous le graphe, la terminologie « file » sera expliquée plus tard) : a d'abord, d ensuite, e enfin (par exemple dans l'ordre alphabétique, mais cet ordre n'est pas important). On obtient donc le résultat suivant :



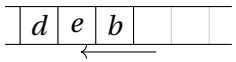
sommets visités : `['c']`



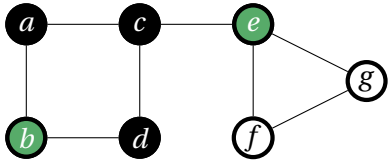
Il faut ensuite recommencer le parcours depuis l'un des sommets en gris clair (l'un des sommets de la file). On recommence avec le sommet a à gauche (c'est-à-dire le premier sommet que l'on a ajouté dans la file) et on le marque comme *visité*. On ajoute alors dans la file les voisins de a **qui n'ont pas encore été visités** et **non encore présents dans la file** : seul b est ajouté.



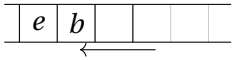
sommets visités : `['c','a']`



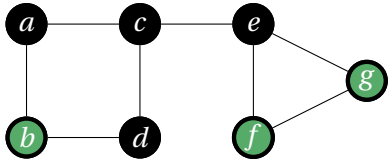
On recommence avec le sommet d à gauche, il est marqué comme visité. Rien n'est ajouté dans la file, puisque b l'était déjà.



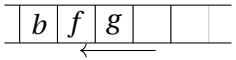
sommets visités : `['c','a','d']`



On recommence avec le sommet e à gauche, il est marqué comme visité. Les sommets f et g sont placés à leur tour dans la file.

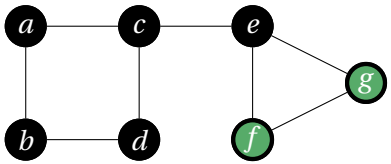


sommets visités : `['c','a','d','e']`

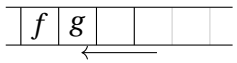


C'est à l'étape précédente que l'on voit la nécessité de choisir le sommet de gauche dans la file ; `e` est à plus petite distance de `c` que `b`

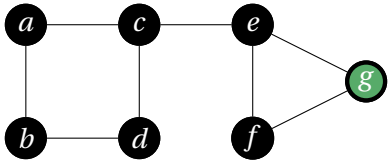
On recommence avec le sommet b, il est marqué comme visité. Il n'a plus de sommets voisins non visités, aucun sommet n'est rajouté dans la file.



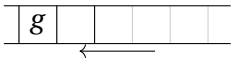
sommets visités : `['c','a','d','e','b']`



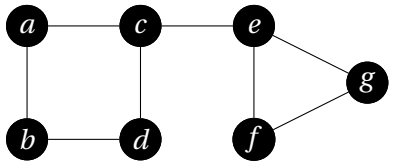
On recommence avec le sommet f, on le marque comme visité, aucun sommet n'est placé dans la file.



sommets visités : `['c','a','d','e','b','f']`



On recommence avec le sommet g, on le marque comme visité. Aucun autre sommet n'est placé dans la fileA



sommets visités : `['c','a','d','e','b','f','g']`

Désormais la file d'attente est vide, le parcours est terminé. À l'issue de l'algorithme, on a un parcours dans un certain ordre de tous les sommets *via* la liste des sommets visités. Ce parcours correspond bien à un parcours en largeur : on a d'abord le sommet à distance 0 de la source (`c`), puis à distance 1 (`a, d, e`), et enfin à distance 2 (`b, f, g`).

STRUCTURES DE DONNÉES UTILISÉES. Analysons à partir de l'exemple précédent les structures de données qui seront utilisées. On a besoin :

- d'une liste `L_v`, initialement vide, contenant tous les sommets que l'on a déjà visités dans le parcours. C'est la liste que l'on renverra à la fin du parcours : elle contiendra alors tous les sommets du graphe $\mathscr{G}$ qui sont accessibles depuis le sommet s.
- La liste `File` sera la file d'attente des sommets (représentée sur le dessin plus haut en-dessous du graphe) que l'on doit encore visiter : les premiers sommets que l'on visitera seront les premiers de la file, et les nouveaux sommets à visiter seront ajoutés en queue de liste (avec `append`). Initialement, la liste `File` contiendra le sommet de départ s.

Note | *C'est pour cette raison que l'on qualifie cette liste de « file » : les sommets sont traités selon la priorité « premier arrivé premier servi », comme une file d'attente dans la vie courante.*

- À une étape donnée, on a besoin de tester souvent si un sommet est découvert/visité ou non (c'est-à-dire coloré en gris ou noir, ou non). On peut tout simplement tester l'appartenance à `L_v` et à la file, mais cela est coûteux en temps car nécessite un parcours. Il est préférable d'introduire un dictionnaire `D_dv` de clefs les sommets et valeurs `True` si le sommet a déjà été visité ou découvert, `False` dans le cas contraire. On modifiera donc ce dictionnaire à chaque coloration en noir d'un sommet.

3.2. Implémentation en Python

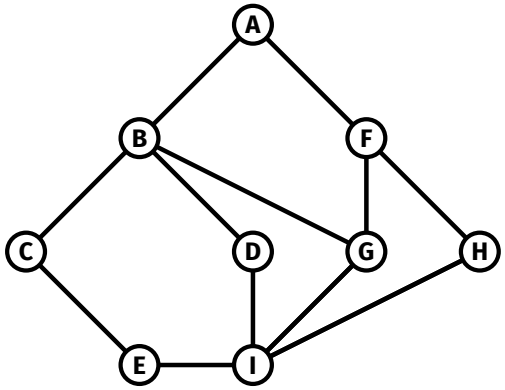
Récapitulons l'algorithme de parcours en largeur présenté dans l'exemple précédent.

(Parcours en largeur) Tant que la liste `File` n'est pas vide, c'est-à-dire tant qu'il reste dans la file d'attente des sommets à explorer, on effectue les actions suivantes :

- on visite le premier sommet (donc celui de gauche sur les dessins précédents) de la file d'attente `File` :
  - on l'ajoute à la liste `L_v` des sommets visités ;
  - dans le dictionnaire `D_dv`, on passe à `True` la valeur du sommet ;
  - on le supprime de la file d'attente `File` ;
- on ajoute à la file d'attente `File` tous les voisins de ce sommet qui ne sont pas déjà visités ou découverts. *(Ainsi, un sommet ne peut pas apparaitre plusieurs fois dans la file)*

Exercice 12 | Étudier l'algorithme ci-dessus et appliquez-le au graphe ci-dessous avec comme sommet de départ le sommet A. Vous noterez à chaque étape :

- les sommets atteints (liste `L_v`)
- les sommets présents dans la file d'attente (liste `File`).



Vous pourrez compléter le tableau ci-dessous.

L_v	File
[]	[A]

Exercice 13 | Code du parcours en largeur

1. Tester et comprendre les commandes ci-après dans une console.

```
>>> L = [1, 2, 3]
>>> x = L.pop(0)
>>> x
>>> L
```

2. Compléter le code ci–dessous :

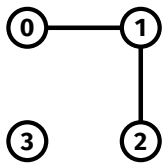
```
def Parcours_largeur(G, s):
    """
    Arguments :
    - un graphe donné sous forme d'un
```

```
    dictionnaire G des voisins ;
    - un sommet s du graphe.
    Sortie : la liste des sommets accessibles depuis le sommet s, dans l'ordre du parcours en largeur.
    """
    D_dv = {}
    for x in G:
        D_dv[x] = ____________

    File = [s]
    D_dv[s] = True

    L_v = []
    while len(File) != ____________ :
        s = ____________ # on récupère le premier sommet de la file et on le supprime
        L_v.append(____________)
        D_dv[s] = ____________
        for v in G[s]:
            if ____________:
                File.append(____________)
                D_dv[v] = ____________
    return ____________
```

3. Tester la fonction sur le graphe G_3.

Exercice 14 | Chemin reliant deux sommets

1. Coder le graphe ci-dessous en Python à l'aide d'un dictionnaire.



Vous pourrez vous servir de ce graphe pour tester les fonctions de cet exercice.

2. Écrire une fonction `existe_chemin(G, s, t)` qui prend en argument un graphe et deux sommets s et t du graphe, et qui détermine s'il existe un chemin du sommet s vers le sommet t dans le graphe. *On pourra donc utiliser un parcours en largeur, puis à la fin du parcours, il suffira d'analyser si t fait partie des sommets visités ou non.*

3. Si cela est possible, on désire alors renvoyer un chemin du sommet s vers le sommet t. Pour cela il nous faut connaitre pour chaque sommet découvert, le « père » (c'est-à-dire le sommet d'où l'on provenait lorsqu'on le marque comme visité).

   3.1) Écrire une nouvelle fonction `parcours_largeur_pred(G, s)` basée sur `parcours_largeur(G, s)` pour quelle renvoie, en plus des sommets visités, le père de chaque sommet découvert. *On pourra stocker cette information dans un dictionnaire `pred` de clefs les sommets, et valeurs le père en question.*

   3.2) Écrire alors une fonction `chemin(G, s, t)` qui renvoie un chemin de s à t. *On indiquera un message lorsqu'aucun chemin n'existe.*

Exercice 15 | Le chou, la chèvre et le loup

Une devinette raconte l'histoire d'un berger qui possède un chou, une chèvre et un loup. En sa présence, la chèvre n'ose pas manger le chou, pas plus que le loup n'ose manger la chèvre, mais ils n'hésiteraient pas à satisfaire leur appétit si l'homme tournait le dos. Ce berger doit traverser une rivière avec sa petite troupe et il ne dispose que d'une barque, dans laquelle il peut naviguer avec un seul de ses compagnons. Comment doit-il s'y prendre pour amener tout le monde sur l'autre rive ? Par exemple, il doit éviter de laisser la chèvre et le loup seuls sur une rive pendant une traversée.

Ce problème peut se modéliser comme un parcours de graphe. Tout d'abord, chacun des quatre protagonistes pouvant se trouver sur une rive ou l'autre, en enlevant les cas dramatiques, il nous reste dix états :

	Rive de départ	Rive d'arrivée
0	–	chou, chèvre, loup, berger
1	chou	chèvre, loup, berger
2	chèvre	chou, loup, berger
3	loup	chou, chèvre, berger
4	chou, loup	chèvre, berger
5	chèvre, berger	chou, loup
6	chou, chèvre, berger	loup
7	chou, loup, berger	chèvre
8	chèvre, loup, berger	chou
9	chou, chèvre, loup, berger	–

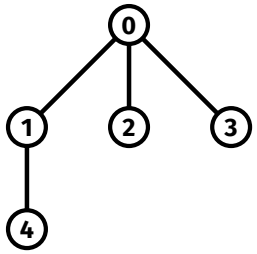
1. Ces dix états numérotés de 0 à 9 constituent les sommets du graphe. Deux sommets sont voisins si l'on peut passer d'un état à l'autre par une traversée en bateau.

Dessiner le graphe correspondant.

2. Résoudre le problème à l'aide d'un parcours de graphe. *On pourra compléter le graphe* `G_berger` *donné dans le fichier externe.*

4. SOLUTIONS DES EXERCICES

Solution (exercice 2) [Énoncé]



Pour le deuxième graphe, on a :

$$S = \{A, B, C, D, E\}$$
$$\mathscr{A} = \{(A, B, 3), (A, C, 2), (A, D, 1), (E, B, 7)\}$$

Solution (exercice 3) [Énoncé]

`E` et `B` sont voisins, alors que `D` et `A` ne le sont pas. En fait `D` est un sommet isolé, il n'est donc voisin de personne. `A` est de degré 3, `B` est de degré 3 aussi, `D` est de degré 0, `E` est de degré 2. Il y a par ailleurs présence d'une boucle en `E`.

Solution (exercice 4) [Énoncé]

1. Pour G_1, la somme des degrés vaut 18 et le nombre d'arêtes 9. Pour G_2, on trouve respectivement 20 et 10, et pour G_3, on trouve 14 et 7. On peut donc conjecturer que la somme des degrés est égale au double du nombre d'arêtes.
2. Chaque arête a deux extrémités. Elle est donc comptée deux fois quand on somme les degrés de tous les sommets.
3. La somme des degrés de tous les sommets d'un graphe est donc un nombre pair.

Solution (exercice 5) [Énoncé]

1. Cela ne semble pas possible pour le premier mais possible pour les trois autres.
2. Empruntons le chemin eulérien, et imaginons que l'on supprime les arêtes qui ont été parcourues. A chaque passage sur un sommet (sauf au début et à la fin), on supprime l'arête qui arrive sur ce sommet et l'arête qui en part. Ainsi, sauf pour le sommet de départ ou d'arrivée, la parité du degré reste inchangée. À la fin du parcours, toutes les arêtes sont supprimées, ce qui permet de conclure sur la parité des sommets.
3. Dans le premier graphe, les quatre sommets sont de degré 3, donc, d'après le théorème d'Euler, il n'existe pas de chemin eulérien. Dans les autres graphes, les degrés des sommets sont respectivement 2,3,3,4,4 (pour le deuxième), 2,2,4,4,4,4 (pour le troisième) et 2,2,3,3,4,4 (pour le quatrième). D'après le théorème d'EULER, il existe donc un chemin eulérien.

Solution (exercice 6) [Énoncé]

```
>>> G_1 = {0 : [1, 2, 3], 1:[0, 4], 2:[0], 3:[0]}
>>> G_2 = {"A" : [("B", 3), ("C", 2), ("D", 1)], "B" : [], "C" \
↪ : [], "D" : [], "E" : [("B", 7)]}
>>> G_1[2]
[0]
>>> [X[0] for X in G_2["A"]]
['B', 'C', 'D']
```

Solution (exercice 7) [Énoncé]

1. Les sommets correspondent simplement aux clefs du dictionnaire.
```
def sommets(G):
    L_s = []
    for s in G:
        L_s.append(s)
    return L_s
```
2. Si un élément est un sommet, alors les voisins sont simplement la valeur associée.
```
def voisins(G, s):
    if s in G:
        return G[s]
```
3. C'est simplement la somme des longueurs des valeurs du dictionnaire.
```
def nb_arcs(G):
    nb_arcs = 0
    for s in G:
        nb_arcs += len(G[s])
    return nb_arcs
```
4. On parcourt les sommets et on note ceux sans voisin.
```
def sans_vois(G):
    L_sans_vois = []
    for s in G:
        if len(G[s]) == 0:
            L_sans_vois.append(s)
```

```
        return L_sans_vois
```

5. Il s'agit ici de vérifier si les arcs $i \to j$ et $j \to i$ sont présents.

```
def est_oriente(G):
    for s in G:
        vois_s = G[s]
        for v in vois_s:
            # on vérifie si s est présent dans les valeurs \
            ↪ de v
            if s not in G[v]:
                return True
    return False
```

6.

```
>>> G_3 = {0 : [1, 6, 7], 1 : [0, 2, 4, 5, 6], 2 : [1], 3 : \
↪ [5, 6, 8], 4 : [1], 5 : [1, 3, 6], 6 : [0, 1, 3, 5, 7], 7 \
↪ : [0, 6, 8], 8 : [3, 7]}
>>> sommets(G_3)
[0, 1, 2, 3, 4, 5, 6, 7, 8]
>>> voisins(G_3, 1)
[0, 2, 4, 5, 6]
>>> voisins(G_3, 9)
>>> nb_arcs(G_3) # le double du nombre d'arêtes non-orientées
26
>>> est_oriente(G_3)
False
>>> D = {0:[1,3], 1:[2,3], 2:[3], 3:[1]}
>>> est_oriente(D) # exemple non orienté
True
>>> sans_vois(G_3) # aucun sommet isolé
[]
```

Solution (exercice 8) [Énoncé]

1.

```
def sommets_m(G):
    N = len(G)
    return list(range(N))
```

2.

```
def voisins_m(G, s):
    L_vois = []
    N = len(G)
    # on parcourt ensuite la ligne s de la matrice
    if 0 <= s < N:
        for t in range(N):
            if G[s, t] == 1:
                L_vois.append(t)
    return L_vois
```

3. C'est simplement le nombre de 1 dans la matrice.

```
def nb_arcs_m(G):
    nb_arcs = 0
    N = len(G)
    for i in range(N):
        for j in range(N):
            if G[i, j] == 1:
                nb_arcs += 1
    return nb_arcs
```

4. On parcourt la ligne i, si elle ne comporte que des 0 alors le sommet i n'a aucun voisin.

```
def sans_vois_m(G):
    L_sans_vois = []
    N = len(G)
    for i in range(N):
        if np.sum(G[i]) == 0:
            L_sans_vois.append(i)
    return L_sans_vois
```

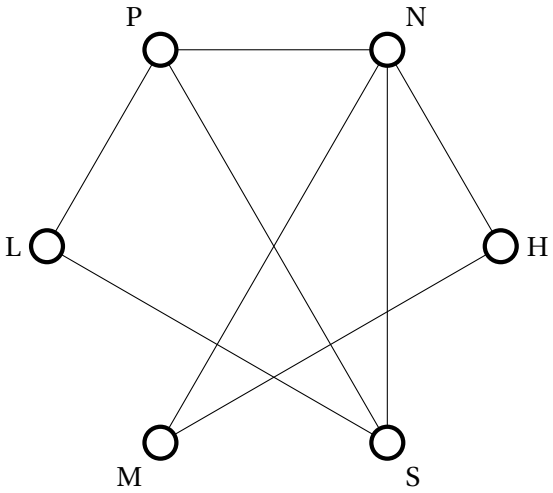
5.

```
>>> G_3m = np.array([[0,  1,  0,  0,  0,  0,  1,  1,  0],
...     [1,  0,  1,  0,  1,  1,  1,  0,  0],
...     [0,  1,  0,  0,  0,  0,  0,  0,  0],
...     [0,  0,  0,  0,  0,  1,  1,  0,  1],
...     [0,  1,  0,  0,  0,  0,  0,  0,  0],
...     [0,  1,  0,  1,  0,  0,  1,  0,  0],
...     [1,  1,  0,  1,  0,  1,  0,  1,  0],
...     [1,  0,  0,  0,  0,  0,  1,  0,  1],
...     [0,  0,  0,  1,  0,  0,  0,  1,  0]])
>>> sommets_m(G_3m)
[0, 1, 2, 3, 4, 5, 6, 7, 8]
>>> voisins_m(G_3m, 1)
[0, 2, 4, 5, 6]
>>> voisins_m(G_3m, 9)
>>> nb_arcs_m(G_3m) # le double du nombre d'arêtes \
↪ non-orientées
26
```

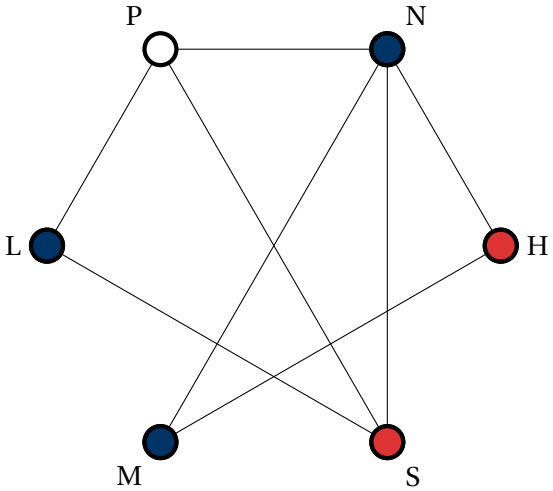
```
>>> sans_vois_m(G_3m) # aucun sommet isolé
[]
```

Solution (exercice 9) [Énoncé]

1. Dessiner le graphe correspondant.



2. Il s'agit ici de donner une coloration possible, le nombre optimal de couleurs (*a priori* inconnu) sera alors inférieur au nombre de couleur trouvé.



On constate ici que 3 couleurs ont été utilisées, *a priori* on ne peut faire mieux puisque les relations entre P, S et L imposent déjà 3 couleurs, donc 3 créneaux.

Solution (exercice 10) [Énoncé]

1.
```
def premierNonPresent(L) :
    """
    renvoie le plus petit entier naturel non présent
    dans la liste d'entiers naturels L
    """
    i = 0
    while i in L :
        i += 1
    return i
```

```
>>> premierNonPresent([0, 2, 3])
1
>>> premierNonPresent([0, 1, 2])
3
```

Ce n'est pas forcément la fonction la plus rapide (lenteur du test **in** *qui nécessite à chaque itération de parcourir toute la liste), on peut faire mieux.*

2.
```
def coloriage(G, S):
    """
    renvoie le dictionnaire correspondant au coloriage des \
    ↪ sommets du graphe G pour le sens de parcours précisé \
    ↪ dans S
    """
    C = {}
    for s in S :
        U = []
        for vois in G[s]:
            if vois in C:
                U.append(C[vois])
        C[s] = premierNonPresent(U)
    return C
```

```
>>> coloriage(G, S1) # 4 couleurs utilisées
{'HF': 0, 'N': 1, 'IF': 2, 'GE': 1, 'B': 0, 'PL': 2, 'CVL': \
↪ 0, 'BFC': 3, 'NA': 1, 'ARA': 2, 'O': 0, 'PACA': 1}
>>> coloriage(G, S2) # 5 couleurs utilisées
{'ARA': 0, 'BFC': 1, 'B': 0, 'CVL': 2, 'GE': 0, 'HF': 1, 'IF
': 3, 'N': 4, 'NA': 1, 'O': 2, 'PL': 3, 'PACA': 1}
```

Solution (exercice 11) [Énoncé]

Comme expliqué dans le texte, on doit d'abord avoir les sommets à distance 0 de `c` (donc `c`), puis les sommets à distance 1 (donc `a, d, e`), puis à distance 2 (donc `b, f, g`). Au total, l'algorithme devrait renvoyer :

```
[c, a, d, e, b, f, g]
```

Solution (exercice 12) [Énoncé]

L_v	File
[]	[A]
[A]	[B,F]
[A,B]	[F,C,D,G]
[A,B,F]	[C,D,G,H]
[A,B,F,C]	[D,G,H,E]
[A,B,F,C,D]	[G,H,E,I]
[A,B,F,C,D,G]	[H,E,I]
[A,B,F,C,D,G,H]	[E,I]
[A,B,F,C,D,G,H,E]	[I]
[A,B,F,C,D,G,H,E,I]	[]

Remarque : on a choisi ici de ranger dans la file d'attente les voisins dans l'ordre lexicographique, ce choix est fait au moment du codage du dictionnaire des voisins. Le parcours est donc `[A,B,F,C,D,G,H,E,I]`.

Solution (exercice 13) [Énoncé]

1.
```
>>> L = [1, 2, 3]
>>> x = L.pop(0) # suppression du 1er élément + stockage \
↪ dans x
>>> x
1
>>> L
[2, 3]
```

2.
```
def Parcours_largeur(G, s):
    """
    Arguments :
    - un graphe donné sous forme d'un
    dictionnaire G des voisins ;
    - un sommet s du graphe.
    Sortie : la liste des sommets accessibles depuis le \
    ↪ sommet s, dans l'ordre du parcours en largeur.
    """
    D_dv = {}
    for x in G:
        D_dv[x] = False

    File = [s]
    D_dv[s] = True

    L_v = []
    while len(File) != 0:
        s = File.pop(0)
        L_v.append(s)
        for v in G[s]:
            if not D_dv[v]:
                File.append(v)
                D_dv[v] = True
    return L_v

G_3 = {0 : [1, 6, 7], 1 : [0, 2, 4, 5, 6], 2 : [1], 3 : [5, \
↪ 6, 8], 4 : [1], 5 : [1, 3, 6], 6 : [0, 1, 3, 5, 7], 7 : \
↪ [0, 6, 8], 8 : [3, 7]}

>>> Parcours_largeur(G_3, 6)
[6, 0, 1, 3, 5, 7, 2, 4, 8]
```

Solution (exercice 14) [Énoncé]

1.
```
G_test = {0:[1], 1:[0, 2], 2:[1], 3:[]}
```

2. On parcours le graphe et on s'arrête dès que l'on a trouvé t.

```
def existe_chemin(G, s, t):
    """
    Arguments :
    - un graphe donné sous forme d'un
    dictionnaire D d'adjacence ;
    - deux sommets s et t du graphe.
    Sortie : True si il existe un chemin de s vers t
    False si non
```

```
    """
    L_v = Parcours_largeur(G, s)
    return t in L_v
```

```
>>> existe_chemin(G_test, 3, 1)
False
>>> existe_chemin(G_test, 0, 2)
True
```

3. 3.1)

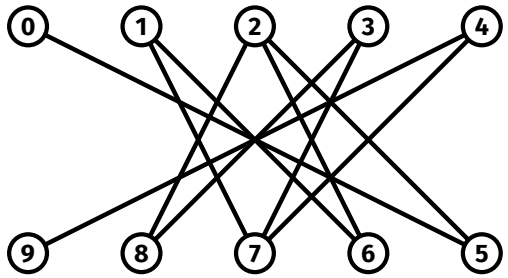
```
def parcours_largeur_pred(G, s):
    """
    Arguments :
    - un graphe donné sous forme d'un dictionnaire des voisins
    - deux sommets s et t du graphe.
    Sortie : les sommets visités et le dictionnaire des prédécesseurs
    """
    n = len(G) # Nombre de sommets du graphe.
    D_dv = {}
    for x in G:
        D_dv[x] = False
    pred = {} # initialisation du dico des prédecesseurs

    File = [s]
    D_dv[s] = True

    L_v = []
    while len(File) != 0:
        s = File.pop(0)
        L_v.append(s)
        D_dv[s] = True
        for v in G[s]:
            if not D_dv[v]:
                File.append(v)
                D_dv[v] = True
                pred[v] = s
    return L_v, pred
```

3.2) Pour reconsituer le chemin, on remonte alors la liste des prédécesseurs jusqu'à revenir à l'arrivée.

```
def chemin(G, s, t):
    if existe_chemin(G, s, t):
        #si le chemin est possible
        L_v, pred = parcours_largeur_pred(G, s)
        C = [t]
        while C[0] != s:
            C = [pred[C[0]]] + C # ajout à gauche de chemin
        return C
```

```
>>> chemin(G_test, 3, 1)
>>> chemin(G_test, 0, 2)
[0, 1, 2]
```

Solution (exercice 15) [Énoncé]



Pour résoudre le problème, il s'agit de savoir s'il existe un chemin menant de l'état 9 (tout le monde est sur la rive de départ), à l'état 0 (tout le monde est sur la rive d'arrivée).

```
G_berger = {0:[5], 1:[6,7], 2:[5,6,8], 3:[7,8], 4:[7,9], \
    5:[0,2],\
    6:[1,2], 7:[1,3,4], 8:[2,3], 9:[4]}
```

On en déduit alors la solution :

```
>>> chemin(G_berger, 9, 0)
[9, 4, 7, 1, 6, 2, 5, 0]
```