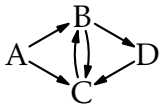
Lycée Les 3 Sources **Bourg-Lès-Valence**
NSI T^{le} – Thème 1 Année **2023-24**

Les graphes

Définition

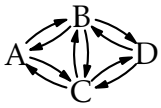
Un **graphe orienté** est un ensemble de **sommets** (ou **nœuds**) reliés par des **arcs**. Dans le graphe suivant, il est possible de passer de A vers B, mais pas de B vers A. Par contre, on peut aller aussi bien de B vers C que de C vers B.

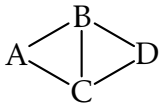


La **taille** d'un graphe est le nombre de sommets qu'il contient.
On appelle **voisins** d'un sommet S l'ensemble des sommets S′ tels qu'il existe un arc allant de S à S′.

Exercice 1 : Déterminer les voisins de chacun des sommets du graphe ci-dessus.

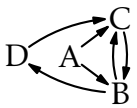
Lorsque pour tout arc, il y a également l'arc réciproque, on dit que le graphe est **non orienté** et dans ce cas, on peut enlever les flèches. On dit alors que ce sont des **arêtes**.

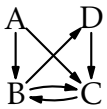




Exercice 2 : Déterminer les voisins de chacun des sommets du graphe non orienté ci-dessus.

Un graphe est entièrement défini par l'ensemble des voisins de chacun de ses sommets.
La représentation n'est qu'un moyen de visualiser cela. Ainsi, la position ou la distance entre les sommets dans le graphe n'a aucune importance. Par exemple, les graphes suivants sont équalents au premier graphe orienté.





Exercice 3 : Représenter le graphe orienté défini par les voisins de chaque sommet :

```
voisins = {"A": ["B"], "B": ["D"], "C": ["A", "B"], "D": ["A"]}
```

Le nombre de voisin d'un sommet s'appelle le **degré** de ce sommet.
Exercice 4 : Déterminer le degré de chaque sommet des exemples de graphes orienté et non orienté ci-dessus.

De la même manière que beaucoup de situations se modélisent en mathématiques par des fonctions ou des (in)équations, un grand nombre de situations en informatique se ramènent à des problèmes de graphes. En voici quelques uns, même si cette liste est loin d'être exhaustive.

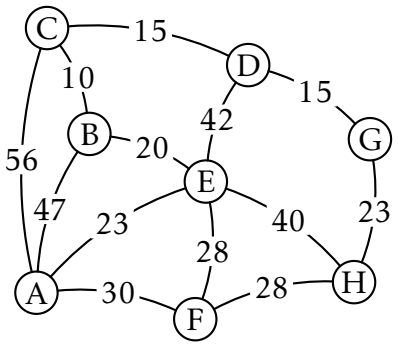
Recherche du plus court chemin

Un **chemin** est une séquence de sommets indiquant les arcs à suivre pour aller du premier sommet au dernier. On dit aussi **chaîne** pour un graphe non orienté. Par exemple, dans le graphe non orienté de la première partie, on peut aller de A à D en suivant le chemin A – B – C – D. En général, on ne passe pas deux fois par la même arête (ou arc) dans un chemin. Ainsi, A – B – C – B – D n'est pas valide car on passe deux fois par l'arête entre B et C. Par contre, dans le graphe orienté, A → B → C → B → D est valide, car on prend deux arcs différents entre B et C.

Exercice 5 : Déterminer tous les chemins allant de A à D dans le graphe non orienté.

La **longueur d'un chemin** est le nombre d'arcs (ou arêtes) qui composent le chemin. Ainsi A – B – D est de longueur 2. Le **plus court chemin** entre deux sommets est un chemin de longueur minimale entre ces deux sommets. Ainsi, entre A et D, A – B – D et A – C – D sont tous les deux des plus courts chemins. Il n'y a donc pas forcément unicité.
La **distance** entre deux sommets est la longueur du plus court chemin les reliant.

Dans les **graphes pondérés**, on rajoute des **poids** sur les arcs. La longueur d'un arc est alors la somme des poids qui le compose.



Exercice 6 : Dans le graphe ci-contre, déterminer le plus court chemin entre A et G.

La recherche du plus court chemin ne se limite pas aux problèmes de GPS. Pour des problèmes d'optimisation, cela permet de trouver comment passer de la configuration initiale à celle désirée en le moins d'étapes possible.

Connexité

Un graphe non orienté est **connexe** si pour tout couple de sommets S et S′, il existe un chemin permettant d'aller de S à S′. Pour un graphe orienté, on dit qu'il est connexe si le graphe non orienté correspondant l'est. Un graphe orienté est **fortement connexe** si pour tout couple S et S′, il y a un chemin allant de S à S′ et un autre permettant d'aller de S′ à S.

Exercice 7 : Déterminer si le graphe orienté de la première page est connexe et fortement connexe.

Exercice 8 : Dessiner tous les graphes connexes non orientés de taille 3.

Recherche de circuits/cycles

On appelle **circuit** un chemin contenant au moins un arc dont le dernier sommet est aussi le premier et ne prenant pas deux fois le même arc. On dit aussi **cycle** pour les graphes non orientés et dans le cas général. La recherche de circuits permet, par exemple, de déterminer s'il est possible de revenir à la configuration initiale. On peut aussi chercher le circuit de taille maximale et en particulier, chercher un cycle passant une fois par chaque sommets, comme dans le problème du voyageur de commerce. Un tel cycle s'appelle un **cycle Hamiltonien**. S'il existe, on dit que le graphe est **Hamiltonien**.

Exercice 9 : Chercher les circuits/cycles de longueur maximale dans les graphes orienté et non orienté de la première page.

Recherche d'un chemin Eulérien

Un **chemin Eulérien** est un chemin passant exactement une fois par chaque arc du graphe. On dit qu'un graphe est **Eulérien** si un tel chemin existe.
Cela permet, entre autres, de savoir s'il est possible de dessiner le graphe sans lever le stylo ou passer deux fois par un même arc. Le graphe pondéré ci-dessus n'est pas Eulérien.

Exercice 10 : Déterminer si le graphe non orienté de la page précédente est Eulérien.

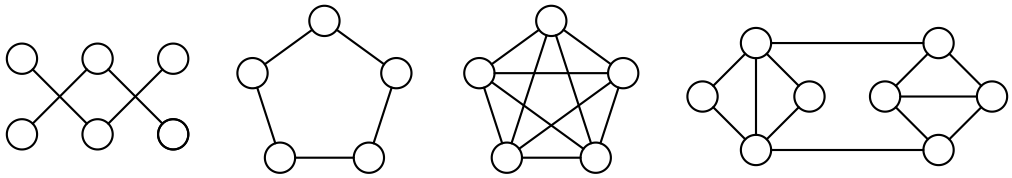
Coloration

Colorer un graphe revient à associer une couleur à chaque sommet, de telle sorte que deux sommets voisins soient forcément de couleurs différentes. Cela permet, par exemple, de déterminer le nombre de groupes qu'il faut faire pour séparer des éléments incompatibles. On appelle **nombre chromatique**, le nombre minimale de couleurs nécessaires pour colorer un graphe. Les problèmes d'emploi du temps ou les sudokus sont également des problèmes de coloration.

Exercice 11 : Déterminer le nombre chromatique du graphe pondéré ci-dessus.

Déterminer le nombre chromatique d'un graphe est un problème difficile. De la même manière, déterminer s'il est possible de colorer un graphe avec k couleurs est difficile si $k > 2$. Il n'y a que le cas $k = 2$ qui est, au contraire, trivial.
Dans certains cas particuliers, on sait déterminer une borne supérieure pour le nombre chromatique. Par exemple, pour des graphes **planaires**, c'est-à-dire qu'on peut dessiner sans que les arêtes se croisent, on sait qu'il faut au plus 4 couleurs. C'est le théorème des 4 couleurs, qui est applicable pour toutes les cartes géographiques.

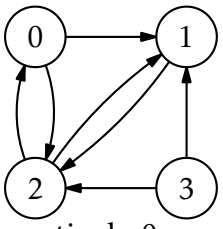
Exercice 12 : Déterminer le nombre chromatique des graphes suivants :



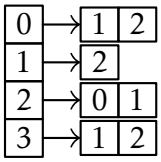
Représentation des graphes

Il existe plusieurs représentations pour les graphes. Le plus souvent, on utilise un schéma, ce qui permet de visualiser facilement les chemins possibles. On peut également utiliser une **matrice d'adjacence** ou des **listes de successeurs**.
La matrice d'adjacence est une matrice carrée dont chaque ligne et chaque colonne correspond à un sommet. Il y a un 1 sur la ligne du sommet i et de la colonne j si et seulement s'il y a un arc reliant i à j.

Les listes de successeurs associent à chaque sommet i la liste, ou l'ensemble, des sommets j tels qu'il y a un arc entre i et j. Voici les 3 représentations d'un même graphe.



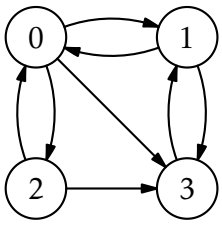
$$\begin{array}{c|cccc} \curvearrowright & 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 2 & 1 & 1 & 0 & 0 \\ 3 & 0 & 1 & 1 & 0 \end{array}$$



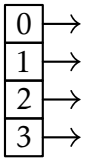
Lorsque les sommets sont numérotés à partir de 0, on peut utiliser une liste de listes. Dans l'exemple ci-dessus, on obtient alors `[[1, 2], [2], [0, 1], [1, 2]]`.

Exercice 13 : Compléter les représentations de chacun des graphes suivants :

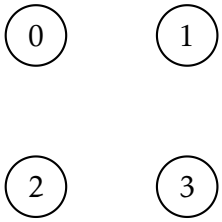
1)



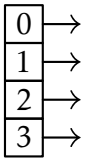
$$\begin{array}{c|cccc} \curvearrowright & 0 & 1 & 2 & 3 \\ \hline 0 & & & & \\ 1 & & & & \\ 2 & & & & \\ 3 & & & & \end{array}$$



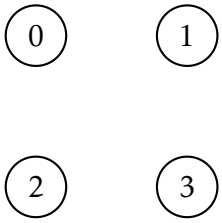
2)



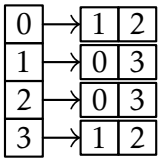
$$\begin{array}{c|cccc} \curvearrowright & 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 1 \\ 3 & 1 & 1 & 0 & 0 \end{array}$$



3)



$$\begin{array}{c|cccc} \curvearrowright & 0 & 1 & 2 & 3 \\ \hline 0 & & & & \\ 1 & & & & \\ 2 & & & & \\ 3 & & & & \end{array}$$



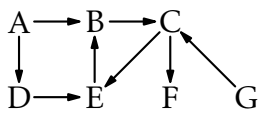
Parcours de graphes

Un graphe peut, tout comme un arbre, être parcouru en largeur ou en profondeur. La différence principale avec les arbres, c'est qu'il faut choisir le sommet de départ.
Le principe est le même pour les deux parcours :

- On place le sommet de départ dans une structure (file pour un parcours en profondeur et pile pour un parcours en largeur).
- Tant que la structure n'est pas vide, on retire un sommet.
- On parcourt chacun de ses voisins qui n'ont pas encore été vus, on les ajoute à la structure et on les marque comme vus.
- Une fois qu'on a fini de parcourir tous les voisins du sommet, on le note comme traité.
- On passe alors au sommet suivant.

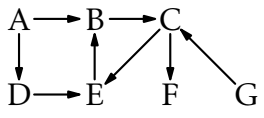
On remarque que les sommets peuvent être dans 3 états différents qui correspondent à "pas encore traité" (`"pas vu"`), "en cours de traitement" (`"vu"`) ou "traité" (`"traité"`).
On utilise parfois des sytèmes avec 3 valeurs ou couleurs qu'on associe à chaque sommet pour déterminer son état. La distinction entre les deux derniers états n'est pas forcément nécessaire.

Exercice 14 : On considère la fonction ci-contre. Déterminer l'évolution de la file d'attente et le résultat obtenu par `parcours_largeur(g, 'A')` à partir du graphe ci-dessous. On supposera que les voisins seront toujours parcourus dans l'ordre alphabétique.




```python
def parcours_largeur(graphe, depart):
    file_attente = file_vide()
    etat = {s: "pas vu" for s in sommets(graphe)}
    etat[depart] = "vu"
    enfile(depart, file_attente)
    ordre_parcours = []
    while not est_vide(file_attente):
        s = defile(file_attente)
        ordre_parcours.append(s)
        for v in voisins(s, graphe):
            if etat[v] == "pas vu":
                enfile(v, file_attente)
                etat[v] = "vu"
        etat[s] = "traité"
    return ordre_parcours
```

Exercice 15 : On considère la fonction ci-contre. Déterminer l'évolution de la pile d'attente et le résultat obtenu par `parcours_profondeur(g, 'A')` à partir du graphe ci-dessous.




```python
def parcours_profondeur(graphe, depart):
    pile = pile_vide()
    etat = {s: "pas vu" for s in sommets(graphe)}
    etat[depart] = "vu"
    empile(depart, pile)
    ordre_parcours = []
    while not est_vide(pile):
        s = depile(pile)
        ordre_parcours.append(s)
        for v in voisins(s, graphe):
            if etat[v] == "pas vu":
                empile(v, pile)
                etat[v] = "vu"
        etat[s] = "traité"
    return ordre_parcours
```