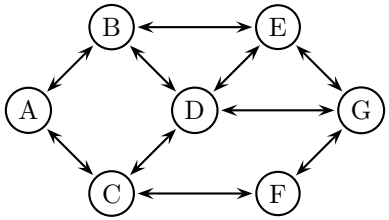
Informatique - Python - TP5
Graphes non pondérés et parcours

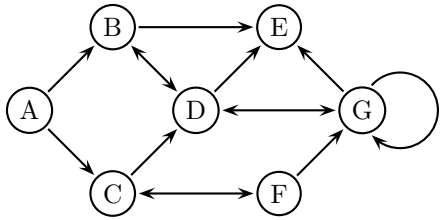
Objectifs :

- Revoir la structure de graphe et ses représentations : matrice d'adjacence, liste (ou dictionnaire) d'adjacence.
- Parcours de graphe et applications : parcours en largeur, (*) en profondeur.

1. Les graphes : vocabulaire et représentations en machine



Graphe 1 : non pondéré, non orienté



Graphe 2 : non pondéré, orienté

- Un graphe est un ensemble de **sommets** ET un ensemble d'**arêtes**. Une arête est un couple de sommets schématisé par un segment fléché. Par exemple, dans le graphe 2, l'arête (B, E) existe (flèche de B vers E), mais pas l'arête (E, B).
  Le nombre de sommets est l'**ordre** du graphe.
- Pour un sommet X donné, les sommets Y pour lesquels il existe une arête (X,Y) sont dits **adjacents** à X et sont appelés les **voisins** de X.
  Attention, si le graphe est orienté ce n'est pas forcément une relation symétrique : dans le graphe 2, B est un voisin de A mais A n'est pas un voisin de B.
  → **Quel est l'ordre du graphe 2 ? Quels sont les voisins de B dans ce graphe ?**
- Un graphe peut être représenté par une **matrice d'adjacence** : en numérotant les sommets de 0 à $n-1$, le terme d'indices $(i;j)$ [1] de la matrice (ligne i, colonne j) est le nombre d'arêtes (sommet i, sommet j) (0 s'il n'y en a pas). Dans nos graphes, il y a au plus une arête reliant 2 sommets, donc la matrice ne contient que des 0 et des 1.
  Ex. La matrice d'adjacence du graphe 1 est :

$$M1 = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 \end{pmatrix}$$

  On remarque qu'elle est symétrique, car le graphe 1 n'est pas orienté.
  → **Écrire la matrice d'adjacence** $M2$ du graphe 2. Est-elle symétrique ? Pourquoi ?

1. Pour simplifier le passage à la programmation, on numérote les lignes et les colonnes de 0 à $n-1$.

- Lorsque le graphe a beaucoup de sommets, le stockage de la matrice d'adjacence prend de la place en mémoire ; de plus, s'il y a peu d'arêtes, la matrice est "creuse" (beaucoup de 0). Pour utiliser plus efficacement la mémoire, on préfère dans ce cas une représentation plus directe : la **liste d'adjacence**, une liste des listes des voisins de chaque sommet.
  Une variante équivalente mais plus "lisible" est le **dictionnaire d'adjacence** : les clés sont les sommets, la valeur associée à chaque sommet est la liste de ses voisins.
  Par exemple, le dictionnaire d'adjacence du graphe 1 est {'A' :['B','C'], 'B' :['A','D','E'], 'C' :['A','D','F'],'D' :['B','C','E','G'],'E' :['B','D','G'],'F' :['C','G'],'G' :['D
  → **Écrire le dictionnaire d'adjacence** du graphe 2.
- Un **chemin** (ou **chaîne**) est une succession de sommets, telle qu'il existe une arête entre chaque sommet et le suivant. Par exemple, A,B,D,G,E est un chemin dans le graphe 2, mais B,D,G,F n'en est pas un (il n'y a pas d'arête (G,F))
- la longueur d'un chemin est le nombre d'arêtes parcourues (le nombre de sommets - 1) : le chemin A,B,D,G,E est de longueur 4.
  La distance entre deux sommets est la longueur d'un plus court chemin existant entre ces deux sommets : par exemple dans le graphe 2, la distance entre A et E est 2 (le chemin A,B,E est de longueur 2 et il n'existe pas de chemin plus court entre A et E)

2. Utilisation de la matrice d'adjacence

- **Préambule** : récupérez et **sauvegardez sous un autre nom** le fichier `TP5_graphes.py` joint, qui sera à compléter. Il contient les matrices d'adjacence `M1` et `M2` des graphes ci-dessus, et le dictionnaire d'adjacence d'un labyrinthe.
- **Puissances de la matrice d'adjacence**
  → **Par une multiplication "à la main"**, obtenir la ligne 3 (c'est-à-dire la quatrième), et seulement celle-là, de la matrice $M1^2$ (carré de la matrice $M1$).

  Vous avez dû obtenir 2 pour le coefficient (3,0) de $M1^2$: c'est le résultat de $M1_{3,1} \times M1_{1,0} + M1_{3,2} \times M1_{2,0}$ (les autres termes de l'addition sont nuls).

  Interprétation : il y a une arête (D,B) et une arête (B,A), donc un chemin D,B,A ; il y a une arête (D,C) et une arête (C,A), donc un chemin D,C,A. Il y a en tout deux **chemins de longueur 2** entre les sommets D et A.
  Résultat général et ébauche de preuve :
  Soit M la matrice d'adjacence d'un graphe d'ordre n. Le coefficient $M^2_{i,j}$ est :

$$\begin{aligned} M^2_{i,j} &= \sum_{k=0}^{n-1} M_{i,k} M_{k,j} \\ &= \sum_{k=0}^{n-1} (\text{nombre d'arêtes } (i,k) \times (\text{nombre d'arêtes } (k,j) \\ &= \sum_{k=0}^{n-1} \text{nombre de chemins } (i,k,j) \end{aligned}$$

  Donc le coefficient d'indices (i,j) de la matrice M^2 donne le nombre de chemins de longueur 2 entre les sommets i et j.

De façon générale, on démontre (par récurrence) que le coefficient (i, j) de la matrice M^a est **le nombre de chemins de longueur a** du sommet i vers le sommet j.

→ **Par une multiplication à la main**, à partir de la ligne 3 de la matrice $M1^2$ obtenue précédemment, obtenir la ligne 3 de la matrice $M1^3$.

Combien y a-t-il de chemins de longueur 3 entre D et A ? Entre D et B ? Vérifiez.

→ **Écrivez une fonction python** nombre_chemins(M,i,j,a) qui renvoie le nombre de chemins de longueur exactement a entre les sommets i et j.

On utilisera la fonction matrix_power du module numpy.linalg : np.linalg.matrix_power(A,k) renvoie la puissance k de la matrice (carrée) A.

Vérifiez sur les exemples précédents.

► **Obtention du dictionnaire d'adjacence à partir de la matrice d'adjacence**

→ **Écrivez une fonction python** dic_adjacence(M) qui renvoie le dictionnaire d'adjacence du graphe dont la matrice d'adjacence est M.

Aide : on crée un dictionnaire vide, puis à chaque sommet on associe la liste vide [].

Il suffit ensuite de "remplir" (append) la liste des voisins de chaque sommet i, en lisant la ligne i de la matrice d'adjacence.

On peut garder des numéros pour désigner les sommets, ou (plus agréable) transformer ces numéros en lettres : en python chr(65+i) donne la lettre correspondant au numéro i ('A' pour 0, 'B' pour 1, etc.)

3. Révision : parcours en largeur

► En partant d'un sommet donné, il s'agit d'explorer d'abord tous les voisins immédiats de ce sommet, **puis** leurs voisins (**s'ils n'ont pas déjà été visités**), etc.

On explore donc d'abord les sommets qui sont à la distance 1 du sommet de départ, puis à la distance 2,...

► C'est l'algorithme qu'on utilise classiquement pour **trouver un plus court chemin** entre deux sommets d'un graphe non pondéré.

Exemple : le parcours complet en largeur du graphe 1 à partir du sommet A est

A (*distance 0*), **B,C** (*distance 1*), **D,E,F** (*distance 2*), **G** (*distance 3*)

Ici par exemple, cet algorithme donne la réponse à la question "Dans le graphe 1, quelle est la distance entre A et F ?" (c'est 2). Dans ce cas, on arrêterait le parcours dès qu'on rencontre le sommet F.

► Dans ce parcours, on récupérera bien sûr les voisins du sommet en cours d'exploration (à la distance distance d), mais ceux-ci devront **attendre** que tous les sommets à la distance d aient été traités avant d'être traités à leur tour !

On utilise pour cela une structure de **file** afin de gérer l'ordre de traitement des sommets.

Dans une file, les éléments sont stockés (***enfilés***) dans leur ordre d'arrivée, et déstockés (***défilés***) également dans leur ordre d'arrivée (principe FIFO : "first in, first out" ou "premier arrivé, premier servi - comme dans la file d'attente à un guichet)

En python, on pourra utiliser une simple liste : les sommets seront enfilés en fin de liste (en queue de file) et défilés en début de liste (en tête de file).

► **Brouillon de l'algorithme :**

```
- créer une file vide
- enfiler le sommet de départ, le marquer comme visité
Tant que la file n'est pas vide et qu'on n'a pas rencontré l'arrivée :
    - défiler le sommet en tête de file, le marquer comme visité
    - enfiler ses voisins (sauf s'ils sont marqués comme visités)
Fin tant que
```

► Précisons la mise en oeuvre :

- Pour savoir si un sommet a déjà été visité, on peut maintenir à jour un dictionnaire **visité** dont les clés sont les sommets, et les valeurs des booléens. Initialement toutes les valeurs sont à False (aucun sommet n'a encore été visité)
- En pratique, on veut pouvoir récupérer un plus court chemin allant d'un sommet à un autre, il faudra donc aussi garder la trace du chemin parcouru. Par exemple, un plus court chemin de A à F est A,C,F (c'est le premier trouvé par l'algorithme). **Il faudra donc, à chaque fois qu'on enfile le voisin d'un sommet X, garder la mémoire du "père" de ce voisin** (c'est le sommet X)
  Pour cela, on peut utiliser un autre dictionnaire **pere**. Ses clés seront les sommets ; initialement personne n'a de père (toutes les valeurs sont initialisées à None), et les valeurs sont mises à jour au fur et à mesure du parcours.
  On pourra ainsi "remonter" la chaîne des pères à la fin (en partant du sommet d'arrivée) et reconstituer le plus court chemin trouvé.

► **Algorithme final : plus_court_chemin(D,depart, arrivee)**

(on suppose que depart ≠ arrivee)

```
Initialiser le dictionnaire pere : tout sommet a pour père None
Initialiser le dictionnaire visité (tout sommet a pour valeur False)
Initialiser la file d'attente à []
Enfiler le sommet depart et le marquer comme visité.
Mettre le booléen fini à False
Tant que la file n'est pas vide et que fini est False :
 X <- defiler la file
 Pour chaque voisin Y de X, si Y n'a pas encore été visité :
  Enfiler Y, le marquer comme visité et mettre à jour son père (c'est X)
  Si Y est le sommet d'arrivée: mettre fini à True
Fin Tant que
Si fini vaut True :
 remonter la chaîne des pères à partir de arrivee jusqu'à obtenir depart,
 renvoyer la liste correspondant au chemin obtenu
Sinon :
renvoyer []    (pas de chemin existant)
```

→ Faire fonctionner "à la main" cet algorithme pour trouver un plus court chemin de B à F dans le graphe 1, en utilisant la file et les dictionnaires **père** et **visité** ci-après (à mettre à jour au fur et à mesure - dans la file on barre les éléments de tête pour les défiler)

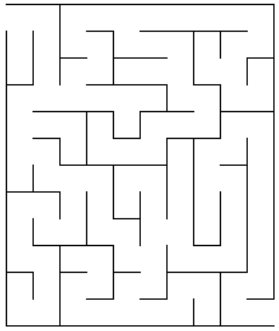
père	**A**	**B**	**C**	**D**	**E**	**F**	**G**
	None	None	None	None	None	None	None

visité	**A**	**B**	**C**	**D**	**E**	**F**	**G**
	False	False	False	False	False	False	False

file							

→ **Programmer cette fonction en python.**

Vérifier sur les graphes "simples" 1 et 2, puis trouver le plus court chemin permettant de sortir du labyrinthe ci-dessous :



Dans ce labyrinthe chaque case est un sommet, et une arête relie deux sommets si on peut passer (en un déplacement) du premier au deuxième.

Les cases de la première ligne sont numérotées de 0 à 9 (de gauche à droite), celles de la deuxième ligne de 10 à 19, ... celles de la dernière ligne de 110 à 119. La case de départ est 0, la case d'arrivée 119.

4. (*) Parcours en profondeur

Le parcours en profondeur consiste à partir d'un sommet, à explorer son premier voisin, puis le premier voisin de ce voisin, et ainsi de suite (jusqu'à atteindre le sommet désiré, ou une impasse). **Lorsqu'on a fini d'explorer un chemin, on revient en arrière** pour explorer le deuxième voisin du sommet précédent, etc.

*Ce procédé se prête particulièrement bien à la **programmation récursive**. On l'utilisera par exemple pour trouver s'il existe un chemin entre deux sommets, ou pour trouver tous les chemins existant entre deux sommets.*

Problème : si on applique ce principe au graphe 1 pour trouver les chemins entre A et G, on obtient A-B-A-B-A-B-.... (cycle infini)

Même en réorganisant l'ordre des voisins, on peut aussi "boucler" sur un cycle plus long : A-B-D-C-A-B-D-C-A-...

Il faut donc interdire les cycles dans le parcours, et pour cela veiller à **ne pas repasser par un sommet déjà visité**.

La fonction récursive aura donc pour paramètres :

- le sommet courant (au premier appel, c'est le sommet de départ choisi)

- le sommet d'arrivée choisi, pour savoir où s'arrêter

- le dictionnaire d'adjacence du graphe considéré

- la liste des sommets visités, dans l'ordre, par l'exploration en cours (au premier appel, cette liste contient uniquement le sommet de départ)

→ Écrivez une fonction python `parcours_prof(sommet_courant, arrivee, dico, visites)` **récursive**, qui stockera dans la liste `resultats` (initialisée à [] dans le programme principal) la liste des chemins menant du sommet `depart` au sommet `arrivee`.

Cette fonction ne renvoie rien, mais s'appelle récursivement sur chaque voisin non encore visité du dernier sommet visité, sauf si ce voisin est le sommet d'arrivée (dans ce cas elle stocke dans `resultats` la liste `visites`).

L'élégance de la récursivité consiste à ne pas modifier directement la liste `visites`, mais à transmettre en paramètre à chaque appel récursif la (nouvelle) liste `visites+[voisins]`.

Ainsi, chaque retour d'appel retrouvera la liste `visites` dans l'état où elle était avant l'appel... ce qui permet les "retours en arrière" dans la recherche de tous les chemins.

→ Trouvez tous les chemins de A à E dans le graphe 2, puis tous les chemins permettant de sortir du labyrinthe.