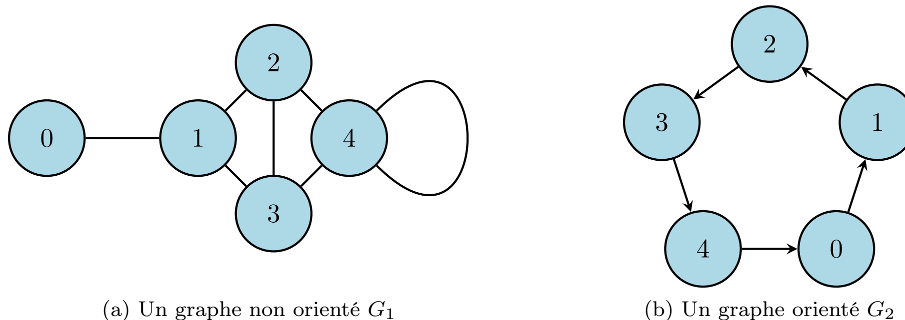
15.1 Qu'est-ce qu'un graphe en informatique ?

En informatique, les graphes sont une structure de données *omniprésentes*, qui permettent de modéliser beaucoup (beaucoup) de situations intéressantes du monde réel. Réseau de communication, réseau routier, relations d'un réseau social, réseau de transport, structures moléculaires... Derrière beaucoup d'objets du quotidien (Google Maps, Twitter, Internet) se cache la théorie des graphes.

Définition

Voici quelques graphes



(a) Un graphe non orienté G_1 (b) Un graphe orienté G_2

FIGURE 15.1 – Premiers exemples de graphes

Ce qu'on appelle *graphe* en informatique est simplement un ensemble fini de *noeuds*, ou sommets, souvent noté V (pour *vertex* en anglais), reliés entre eux par un ensemble fini d'arêtes, souvent noté E (pour *edge* en anglais). On représente une arête par un couple de noeuds (v_1, v_2) où $v_1, v_2 \in V$, correspondant donc aux deux noeuds que relie l'arête. Si les arêtes ont un sens, comme des flèches, alors l'ordre de v_1 et v_2 dans le couple (v_1, v_2) a une importance, on parle alors de *graphe orienté* (voir Figure 15.1b Si les arêtes n'ont pas de sens, l'ordre n'importe pas et on parle de *graphe non orienté* (voir Figure 15.1a). Notez que v_1 et v_2 ne sont pas nécessairement distincts.

Lorsque deux sommets sont reliés par une arête, on dit qu'ils sont *adjacents*, ou *voisins*.

1. Écrire les ensembles V et E décrivant chacun des deux graphes représentés figure 15.1

Ainsi, un graphe G est simplement la donnée d'un couple (V, E), V étant un ensemble fini, et $E \subseteq V \times V$. Ce modèle est très simple mais peut s'adapter à beaucoup de situations constituées d'objets et de relations entre ces objets (ami.e sur Facebook, ligne de métro entre des stations, etc). En mathématiques, on étudie les graphes comme objets en soi, en informatique on tente d'écrire des algorithmes capables d'extraire efficacement des informations d'un graphe. Ce TP introduit quelques uns de ces algorithmes.

Représentation d'un graphe

La structure de graphe n'est pas directement implémentable par une structure `Python` vue jusqu'ici. Ce n'est pas vraiment une liste, ni un dictionnaire par exemple. Pour un graphe à n sommets, on prendra systématiquement la convention que l'ensemble des sommets est $\{0, \dots, n-1\}$, et la difficulté est de représenter les arêtes. Il existe deux conventions :

(i) représentation par la *matrice d'adjacence*, qui est la matrice $M \in \mathcal{M}_n(\mathbb{R})$ telle que

$$\begin{cases} M_{i,j} = 1 & \text{s'il existe une arête de } v_i \text{ vers } v_j, \text{ autrement dit } (v_i, v_j) \in E \\ M_{i,j} = 0 & \text{sinon} \end{cases}$$

(ii) représentation par *listes d'adjacence*, pour laquelle on donne, pour chaque sommet, la liste de ses voisins, c'est à dire qu'on représente G par une liste de liste :

```
G = [
  [...], # liste des voisins du sommet 1
  [...], # liste des voisins du sommet 2
   ...
  [...], # liste des voisins du sommet n
]
```

Notez qu'à partir de chacune de ces reprétentations on peut retrouver n le nombre de sommets. Il n'y donc pas besoin d'autre chose pour représenter G que de sa matrice d'adjacence, ou sa liste d'adjacence.

2. a) Écrire en `Python` les deux représentations de G_1 et G_2 (voir figure 15.1).

b) Écrire deux fonctions `liste_to_matrice(l)` et `matrice_to_liste(M)` qui convertissent entre les deux conventions de représentation d'un graphe. Vérifier qu'on obtient les bons résultats avec les exemples écrits à la main à la question précédente.

3. Comment se traduit sur la matrice d'adjacence le fait qu'un graphe soit non orienté ?

15.2 Premiers algorithmes

Ajout et suppression de noeuds et d'arêtes

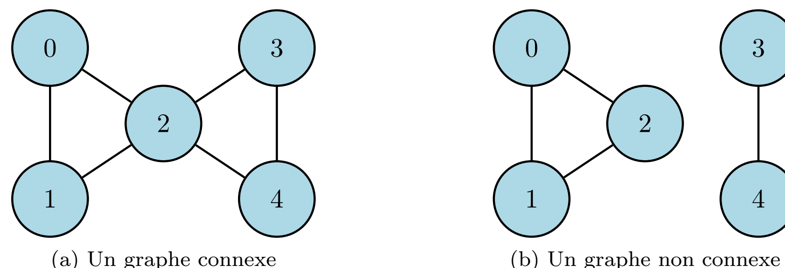
On choisit de travailler ici avec des matrices d'adjacence et des graphes non orientés.

4. a) Écrire deux fonctions `ajoute_noeud(M)` et `supprime_noeud(M, v)` qui renvoie un nouveau graphe avec, respectivement, le noeud n ajouté et le noeud v supprimé.

b) Écrire deux fonctions `ajoute_arete(M, v1, v2)` et `supprime_arete(M, v1, v2)`. Tester les fonctions sur le graphe G_1.

Parcours de graphe

Connexité On parle de *chemin dans le graphe G* lorsqu'on considère une suite de sommets de G $v_1, \dots, v_p$ telle que deux éléments successifs sont reliés par une arête, autrement dit $(v_i, v_{i+1}) \in E$ pour tout i. La détection de chemins dans un graphe, en particulier de plus court chemin, forme tout un pan de la théorie des graphes. Un graphe non orienté est *connexe* si de tout sommet v il existe un chemin à tout autre sommet w. Moins formellement, un graphe est connexe s'il n'est pas en plusieurs morceaux, voir figure 15.2.



(a) Un graphe connexe (b) Un graphe non connexe

FIGURE 15.2 – Connexité d'un graphe

Le graphe des amis Facebook est-il connexe ? Autrement dit, tout le monde est-il l'ami.e d'ami.e (...) d'ami.e de n'importe qui ? Le but de cette question est d'écrire une fonction `est_connexe(M)` qui prend en argument une matrice d'adjacence `M` et renvoie `True` si et seulement si le graphe que représente `M` est connexe. Pour cela, l'idée est de partir d'un sommet arbitraire, par exemple le sommet 0, et de visiter ses voisins, les voisins de ses voisins, les voisins de ses voisins de ses voisins, etc. Si à la fin on a visité tous les noeuds, c'est que le graphe est connexe, sinon le graphe n'est pas connexe. Il faut donc parcourir *récursivement* les voisins des sommets visités.

Rappel sur la récursivité Une fonction Python f a le droit de *faire appel à elle-même*. L'exemple classique est la fonction `factoriel(n)` qui peut s'écrire

```
def factoriel(n):
    if n == 0:
        return 1
    else:
        return n * factoriel(n-1)
```

en utilisant le fait que $n! = n \cdot (n-1)!$.
De manière générale, lorsque l'on écrit une fonction récursive, il y a une partie qui assure la *terminaison*, c'est-à-dire qui s'assure que la fonction va finir par arrêter de s'appeler elle-même, et une partie qui "fait le travail". Ici, le test `if n == 0` est là pour assurer la terminaison. Implicitement, on suppose que l'utilisateur.ice n'utilise `factoriel` qu'avec des entiers `n >= 0`. Que se passe-t-il si on appelle `factoriel(-1)` ? `factoriel(1.5)` ?

5. Écrire de façon récursive une fonction `fibonacci(n)` qui renvoie le n-ième terme de la suite de Fibonacci $0, 1, 1, 2, 3, 5, 8, 13, 21, \dots$.

Ici, on va écrire une fonction `parcours(v, M, dejaVus)` qui parcourt tous le graphe et va s'appeler récursivement sur les voisins des sommets visités. `v` est le noeud en train d'être visité, `M` est la matrice d'adjacence du graphe, et `dejaVus` est un tableau de booléens de taille `n`, qui indique quel sommet a déjà été visité. Ainsi, le graphe est connexe si et seulement si à la fin de la fonction, tous les éléments de `dejaVus` sont `True`.

Attention, si le graphe contient un *cycle*, c'est-à-dire un chemin qui part et arrive au même noeud, il y a un risque que la fonction fasse des appels récursifs infinis. Par exemple s'il y a une arête $(0, 1)$ et une arête $(1, 0)$, appeler aveuglément la fonction `parcours` sur tous les voisins conduit au comportement suivant

(i) je visite 0

(ii) je visite 1 car c'est un voisin de 0

(iii) je visite 0 car c'est un voisin de 1

(iv) je visite 1 car c'est un voisin de 0

(v) je visite 0 car c'est un voisin de 1

(vi) etc

Il faut donc être prudent sur la condition de terminaison !

6. a) Écrire la fonction `parcours(v, M, dejaVus)`.

b) En déduire la fonction `est_connexe(M)`.

c) La tester sur les graphes figure 15.2.

Parcours de graphe, version non récursive

On peut également écrire la fonction `parcours(v, M)` sans récursivité. Cette fois-ci, au lieu d'appeler récursivement la fonction sur les voisins, on va garder dans une variable `aVisiter` les sommets qu'il faut encore visiter (car ce sont les voisins de sommets déjà visités). La pseudo code de la fonction ressemble donc à celui-ci :

Function *Parcours(v, M)*
- àVisiter $\leftarrow \{v\}$
- déjàVus $\leftarrow \{v\}$
- **while** *àVisiter* $\neq \emptyset$ **do**
  - prendre $u \in$ àVisiter
  - **for** w *voisin de* u **do**
    - **if** $w \notin$ *déjàVus* **then**
      - rajouter w à àVisiter
      - rajouter w à déjàVus
    - **end**
  - **end**
- **end**

Le pseudo-code laisse le choix sur la façon de choisir u dans àVisiter. En fait, selon la façon qu'on choisit, on effectue soit un *parcours en profondeur*, soit un *parcours en largeur*.

7. En utilisant le pseudo-code précédent, écrire une fonction `parcoursEnProfondeur(v, M)`. Lorsque l'on choisit u dans àVisiter, on prend le sommet qui a été ajouté le plus récemment. On dit que àVisiter à une structure de *pile* (penser à une pile d'assiette), car elle vérifie la propriété LIFO *Last In First Out*.

8. Écrire cette fois une fonction `parcoursEnLargeur(v, M)`, pour laquelle on choisit u dans àVisiter comme celui qui y a été ajouté en premier. On dit alors que àVisiter à une structure de file (penser à une file d'attente), car elle vérifie la propriéte FIFO *First In First Out*.

9. Tester les fonctions sur le graphe figure 15.3 en partant du sommet 0. Ajouter un `print(w)` au moment où vous ajoutez `w` à déjàVus. Comprenez vous pourquoi on parle de parcours en *profondeur* et en *largeur* ?

10. La version récursive de l'algorithme effectuait-elle un parours en largeur ou un parcours en profondeur ?

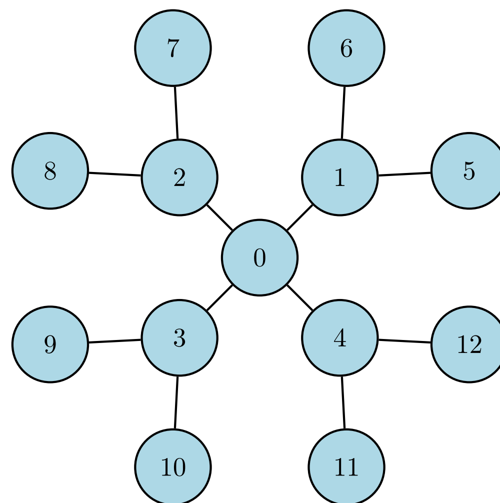


Figure 15.3 – Un graphe en flocon de neige