Cours 4

Chapitre 20 - Graphes : modèle et représentations

De manière générale, un graphe permet de représenter les connexions entre les éléments d'un ensemble, autrement dit tous types de réseaux : des réseaux routiers aux réseaux sociaux, en passant par les réseaux de communications. Le champ d'application des graphes s'est ensuite élargi à de nombreux domaines : mathématiques, chimie, biologie, sciences sociales, etc.

Dans le prochain chapitre, nous nous intéresserons à la conception d'algorithmes pour résoudre des problèmes faisant intervenir des graphes, notamment la recherche de plus courts chemins. Pour ce faire, nous avons besoin dans un premier temps de donner des éléments de vocabulaire précis sur les graphes, puis d'expliquer comment les représenter informatiquement. C'est l'objectif de ce chapitre.

Objectifs

- Se familiariser avec le vocabulaire des graphes, puis le maîtriser ;
- Savoir représenter informatiquement un graphe et maîtriser les manipulations usuelles en Python ;
- Comprendre et savoir implémenter les algorithmes de graphes au programme de prépa, à savoir les parcours en largeur et en profondeur, ainsi que l'algorithme de Dijkstra (pour le prochain chapitre).

1 Vocabulaire sur les graphes

1.1 Généralités

Définition : graphe

Un *graphe* est la donnée d'un ensemble fini S d'éléments appelés les *sommets* du graphe, et d'un ensemble $A \subset S^2$ d'éléments appelés les *arêtes* ou *arcs* du graphe. On note alors le graphe $G = (S, A)$.

L'ensemble A des arêtes correspond aux « connexions » qui existent entre les sommets du graphe représentés par l'ensemble S.

Remarques : autres notations

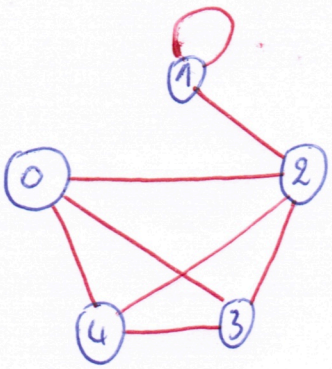
- On trouve bien plus souvent la notation $G = (V, E)$ pour un graphe : V pour **V**ertices (« sommets » en anglais) et E pour **E**dges (« arêtes » en anglais).
- On utilisera donc souvent les lettres u, v, w... pour désigner un sommet d'un graphe, et la lettre e pour désigner une arête.
- On rencontre aussi le terme de *nœud* pour désigner un sommet (mais plus rarement, car il est plutôt réservé aux structures arborescentes).

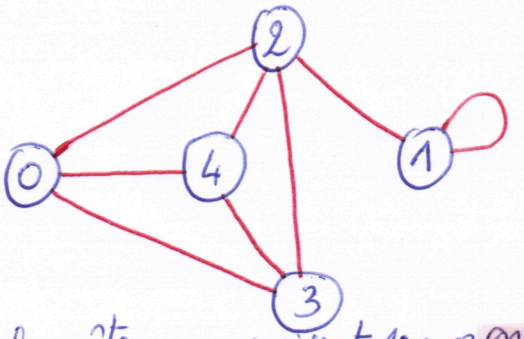
Les graphes tirent leur nom du fait qu'on peut les représenter graphiquement :

- à chaque sommet du graphe, on fait correspondre un point du plan ;
- puis on relie les points correspondant aux extrémités de chaque arête.

En particulier, selon les positions choisies pour placer les différents sommets, on peut obtenir deux représentations graphques différentes d'un même graphe.

Exemple 1. Donner deux représentations graphique du graphe $G = (S, A)$ où $S = \{0, 1, 2, 3, 4\}$ et $A = \{(0,2), (0,3), (0,4), (1,1), (1,2), (2,3), (2,4), (3,4)\}$.





Ici, les arêtes ne se croisent pas donc graphe planaire

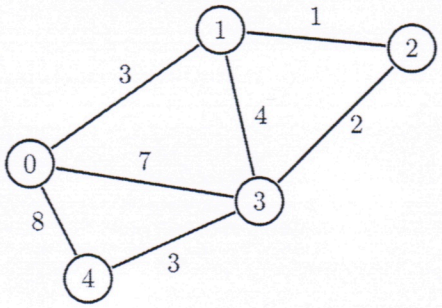
L'écriture précédente de l'ensemble A (mais aussi la représentation graphique proposée) peut sous-entendre que les arêtes peuvent être lues dans les deux sens : si u et v sont connectés, alors v et u le sont forcément aussi... ce n'est hélas pas forcément toujours le cas, par exemple

* réseaux de transports (sens unique, round-point, ...)
* certains réseaux sociaux (on peut "suivre" qqn sans être suivi en retour)

Pour distinguer ces deux situations, on parlera de graphe *orienté* ou de graphe *non orienté* (plus de détails dans les parties 1.2 et 1.3).

Définition : graphe pondéré

Un *graphe pondéré* est la donnée d'un graphe $G = (S, A)$ et d'une fonction $w : A \to \mathbb{R}^+$ définie sur l'ensemble des arêtes de G, appelée *fonction de pondération.*



Autrement dit, un graphe pondéré est simplement un graphe sur lequel les arêtes sont étiquetées par une valeur (en général réelle et positive). Cela peut correspondre par exemple à la distance entre deux sommets dans le cas d'un réseau de transports. Nous reparlerons des graphes pondérés à la fin du chapitre, et surtout lors de l'étude de l'algorithme de Dijkstra. Dans toute la suite, nous considérerons donc (sauf mention contraire) des graphes non pondérés.

1.2 Graphe non orienté

Définitions : graphe non orienté

- Dans un graphe non orienté $G = (S, A)$, une arête $e \in A$ correspond à une paire **non ordonnée** de sommets, que l'on note $e = (u, v)$, avec $u, v \in S$. On considère donc qu'on a à la fois $(u, v) \in A$ et $(v, u) \in A$.
- Si $e = (u, v) \in A$, on dit que u et v sont les *extrémités* de l'arête e, que u et v sont *adjacents* ou *voisins*, et que l'arête e est *incidente* à u et à v.
- Si $u = v$, on dit qu'une arête $(u, u) \in E$ forme une *boucle*.

Dans toute cette sous-partie, on considère $G = (S, A)$ un graphe non-orienté.

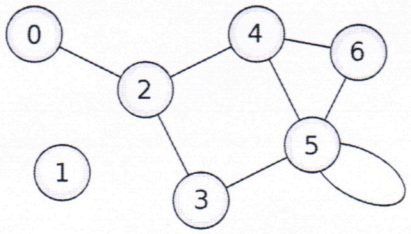
Définition : degré d'un sommet

Soit $u \in S$ un sommet de G. Le *degré* de u, que l'on note en général $d(u)$ ou $\deg(u)$, est le nombre de sommets qui sont adjacents à u.

Cas particulier des boucles

Une boucle de la forme (u, u) contribue deux fois au degré de u (une fois pour chaque extrémité).

Exemple 2. On considère le graphe non orienté représenté ci-dessous :



sommet u	0	1	2	3	4	5	6
degré $d(u)$	1	0	3	2	3	5	2

Lemme des poignées de main

Si $G = (S, A)$ est un graphe non orienté, alors $\sum_{u \in S} d(u) =$ $2|A|$ où $|A|$ = nbre des arêtes.

Démonstration: chaque arête (v, w) est comptée 2 fois :
* une fois dans $d(v)$
* une fois dans $d(w)$

Ce résultat ne semble pas très intéressant de prime abord, mais nous le réutiliserons très souvent dans les analyses de complexité faisant intervenir des graphes.

Définitions : chemin, cycle, distance

Soient u et v deux sommets de G.

- Un *chemin* de u à v est une suite finie $(v_0, v_1, \dots, v_p)$ de sommets telle que
  - $v_0 = u$ et $v_p = v$;
  - $(v_i, v_{i+1}) \in A$ pour tout $i \in [\![0 ; p-1]\!]$.

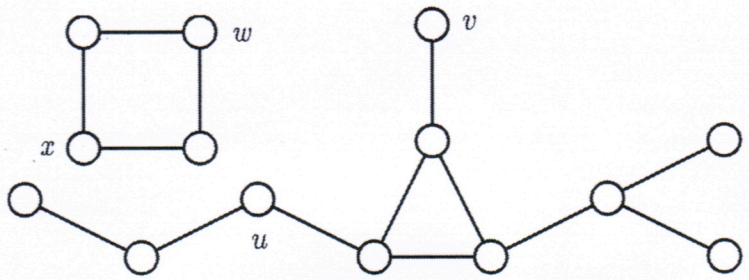
  Le chemin est alors composé de p arêtes : on dit que p est la *longueur* du chemin.

- Si $u = v$, on dit que le chemin précédent est *cyclique*, on parle aussi directement de *cycle*.
- La *distance* entre u et v, notée $d(u, v)$, est la longueur d'**un** plus court chemin entre u et v (il peut y en avoir plusieurs de longueur minimale).

Attention !

Il ne faut pas confondre les cycles et les boucles. Ces dernières sont des arêtes de la forme (u, u), et correspondent donc aux cycles de longueur 1...

Exemple 3. On considère le graphe non orienté ci-dessous.



- $d(u, v) =$ 3 Combien de plus courts chemins de u à v ? 1
- $d(w, x) =$ 2 Combien de plus courts chemins de w à x ? 2
- $d(u, w) =$ ∞ (convention en cas d'inexistance de chemins)

Définitions : connexité

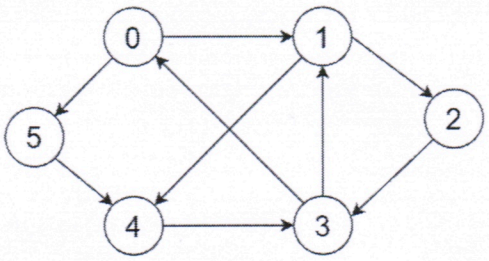
- Un graphe $G = (S, A)$ non orienté est dit *connexe* si pour tout couple de sommets $(u, v) \in S^2$, il existe un chemin de u à v dans G.
- Un graphe non connexe peut être décomposé en plusieurs *composantes connexes* : ce sont les sous-graphes connexes de tailles maximales (c'est-à-dire qui contiennent le plus de sommets).

Par exemple, le graphe précédent contient 2 composantes connexes : le "carré" contenant x et w, et la composante contenant les autres sommets.

1.3 Graphe orienté

Définitions : graphe orienté

- Dans un graphe orienté $G = (S, A)$, un **arc** $e \in A$ correspond à une paire **ordonnée** de sommets, que l'on note toujours $e = (u, v)$, avec $u, v \in S$. La paire (u, v) et la paire (v, u) sont donc deux arcs différents.
- Si $e = (u, v) \in A$, on dit que u est l'*origine* de l'arc e, et que v est son *extrémité*. On dit également que v est un *voisin* de u (attention u n'est pas forcément un voisin de v).



Remarquons que l'on parle désormais d'arcs et non d'arêtes, car ce dernier terme est en général réservé aux graphes non orientés.

Comme les arcs ont cette fois-ci une direction, il nous faut définir deux notions de degré, car un arc peut sortir ou entrer dans un sommet (ou les deux à la fois dans le cas des boucles).

Définitions : degrés entrant et sortant

Soit $G = (S, A)$ un graphe orienté, et soit $u \in S$ un sommet de G. Le *degré entrant* de u, que l'on note $d^-(u)$ ou $\deg^-(u)$, est le nombre d'arcs ayant pour extrémité le sommet u, soit formellement :

$$d^-(u) = \text{Card}(\{v \in S, (v, u) \in A\})$$

On définit de même le *degré sortant* de u, que l'on note $d^+(u)$ ou $\deg^+(u)$, comme étant le nombre d'arcs ayant pour origine le sommet u :

$$d^+(u) = \text{Card}(\{v \in S, (u, v) \in A\})$$

Lemme des poignées de mains (cas orienté)

Si $G = (S, A)$ est un graphe orienté, le lemme des poignées de main devient

$$\sum_{u \in S} d^+(u) = \sum_{u \in S} d^-(u) = |A| \text{ où } |A| = \text{nombre d'arc}$$

chaque arc (v, w) est compté une fois dans chaque somme.

Rq : Il n'y a plus de facteur 2 ici, mais dans les 2 cas, on peut faire : $\sum_{u \in S} d^{(+/-)}(u) = O(|A|)$ (écriture rapide).

Les notions de chemin, cycle et distance ne changent pas quand on passe aux graphes orientés. En revanche, la « connexité » pose plus de problèmes car on peut très bien avoir, dans un graphe non orienté, $d(u,v) = +\infty$ mais $d(v,u) < +\infty$

ex : 

On parle alors de *forte connexité* et donc de *composantes fortement connexes*, ces deux notions étant hors programme en ITC.

2 Représentation informatique des graphes

Il existe deux méthodes principales pour représenter un graphe en Python (ou dans tout autre langage de programmation) : les matrices d'adjacence et les listes d'adjacences. Dans les deux cas, **on suppose dans toute la suite que les sommets sont numérotés par des entiers à partir de 0**, c'est-à-dire que $S = \{0, 1, \dots, n-1\}$, où n désigne le nombre de sommets du graphe. Vous verrez l'an prochain comment se dispenser de cette contrainte, par exemple en utilisant un dictionnaire pour associer un sommet à son numéro.

2.1 Matrice d'adjacence

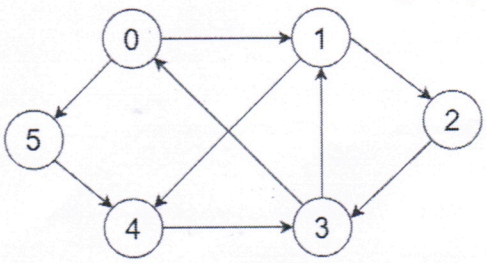
Définition : matrice d'adjacence

Soit $G = (S, A)$ un graphe (orienté ou non) comportant n sommets (numérotés de 0 à $n-1$). La *matrice d'adjacence* de G est la matrice carrée de taille n notée $M = (m_{i,j})_{0 \leq i,j \leq n-1}$ définie par

$$\forall (i,j) \in [\![0\,;\, n-1]\!]^2 \qquad m_{i,j} = \begin{cases} 1 \text{ si } (i,j) \in A \\ 0 \text{ sinon} \end{cases}$$

On peut alors représenter une matrice d'adjacence en Python par un tableau Numpy à 2 dimensions ou une liste de liste (d'entiers ou de booléens).

Exemple 4. On reprend le graphe orienté représenté précédemment. Donner la représentation par matrice d'adjacence de ce graphe sous la forme d'une liste de listes en Python.



n = 6

```
g = [ [0,1,0,0,0,1],   (voisins de 0)
      [0,0,2,0,1,0],   (voisins de 1)
      [0,0,0,1,0,0],   " 2
      [1,1,0,0,0,0],   " 3
      [0,0,0,1,0,0],   " 4
      [0,0,0,0,1,0] ]  " 5
```

Rq : si G est non orienté, la matrice d'adjacence est symétrique.

2.2 Listes d'adjacence

Définition : listes d'adjacence

Soit $G = (S, A)$ un graphe (orienté ou non) comportant n sommets. Une représentation par *listes d'adjacences* de G est la donnée d'une liste `adj` de taille n, telle que pour tout sommet $i \in [\![0\,;\,n-1]\!]$, `adj[i]` contient la liste des sommets voisins de i.

Il s'agit donc également d'une liste de listes, mais celles-ci n'ont plus une taille fixe égale à n : on stocke uniquement les sommets « intéressants », à savoir les voisins.

Exemple 5. Donner la représentation en Python du graphe précédent par listes d'adjacences.

`adj = [[1,5],[4,2],[3],[0,1],[3],[4]]`

Rq : * aucune contrainte d'ordre $[4,2] = [2,4]$
* si $d(u) = 0 \Rightarrow$ `adj[u] = []`.

2.3 Pourquoi deux représentations ?

Chacune des représentations possède ses avantages et ses inconvénients, résumés dans le tableau ci-dessous.

Critère	Matrice d'adjacence	Listes d'adjacences
Quantité de mémoire nécessaire pour stocker le graphe	$O(\lvert S\rvert^2)$ (nombre d'élts de la matrice)	$O(\sum_{u \in S} 1 + d(u)) = O(\lvert S\rvert + \sum d(u))$ ↑ cas d'une liste vide $= O(\lvert S\rvert + \lvert A\rvert)$ (poignées de mains)

Récupérer la liste des voisins d'un sommet u (par exemple pour les parcourir ensuite)	`def voisins(u, M):`
	`V = []`
	`for v in range(len(M)):`
	`if M[u][v] == 1:`
	`V.append(v)`
	`return V`
	Rq : $O(\text{len}(M)) = O(\lvert S\rvert)$
		`def voisins(u, adj):`
		`return adj[u]`
		Rq : $O(\lvert \text{adj}[u]\rvert) = O(d(u))$

Savoir si $(u, v) \in A$, étant donnés u et v	`def est_une_arete(u, v, M):`
	`return M[u][v] == 1`
	Rq : $O(1)$
		`def est_une_arete(u, v, adj):`
		`for w in adj[u]:`
		`if w == v`
		`return True`
		`return False`
		ou `def est_une_arete(u, v, adj):`
		`return v in adj[u]`
		Rq : $O(\lvert \text{adj}[u]\rvert) = O(d(u))$.

listes d'adj plus intéressantes

matrices d'adj plus intéressantes

En résumé : Tout dépend de l'opération principale à effectuer. Dans les algorithmes au programme en ITC, nous verrons que la représentation par listes d'adjacences sera à chaque fois la plus efficace, mais ce n'est pas toujours le cas.

2.4 Et pour les graphes pondérés ?

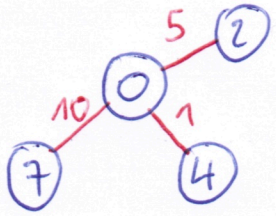
On peut facilement adapter les définitions précédentes pour les graphes pondérés : on rappelle qu'un graphe pondéré est un graphe $G = (S, A)$ muni d'une fonction de pondération $w : A \to \mathbb{R}^+$ sur les arêtes/arcs.

- La matrice d'adjacence $W = (W_{i,j})_{0 \leq i,j \leq n-1}$ associée à un tel graphe peut être définie par

$$\forall (i,j) \in [\![0\,;n-1]\!]^2 \qquad w_{i,j} = \begin{cases} w((i,j)) & \text{si } (i,j) \in A \\ 0 & \text{si } i=j \text{ (pas de boucles)} \\ +\infty & \text{sinon} \end{cases}$$

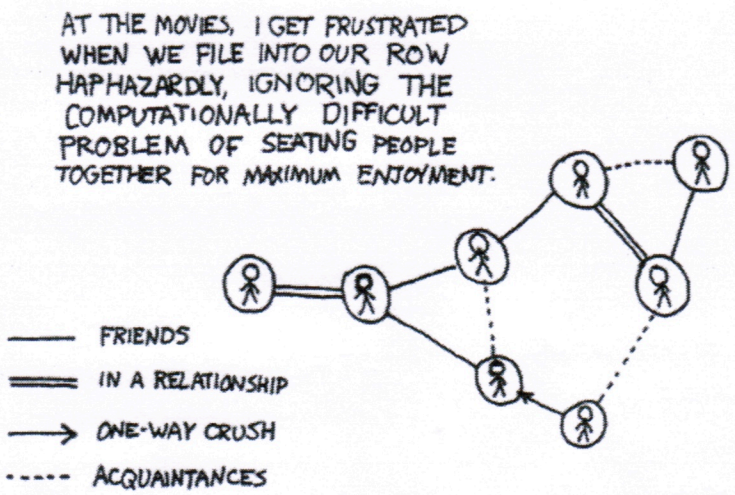
- Une représentation par *listes d'adjacences* d'un graphe G pondéré est la donnée d'une liste `adj` de taille n, telle que pour tout sommet $i \in [\![0\,;n-1]\!]$, `adj[`i`]` est une liste de **couples** de la forme $(v, w((u,v))$, où les sommets v correspondent aux voisins de u.

ex :



adj = [[(2,5), (7,10), (4,1),
⋮
]

Une dernière application des graphes...



Chapitre 4 - Graphes et parcours : rappels et compléments

Ce chapitre reprend l'essentiel des bases de MPSI sur les graphes, en allant un peu plus loin sur les parcours.

Objectifs concours

- Connaître et savoir utiliser le vocabulaire usuel des graphes ;
- Maîtriser les représentations et manipulations usuelles de graphes en Python ;
- Comprendre et savoir implémenter les algorithmes de graphes au programme de prépa, à savoir les parcours en largeur et en profondeur, ainsi que l'algorithme de Dijkstra (au chapitre 4).
- Même si ces algorithmes sont difficiles à implémenter ex nihilo, les concours peuvent considérer qu'il s'agit d'une question de cours car figurant au programme d'ITC. Attention donc !

1 - Rappels

Cette partie comporte principalement des rappels du chapitre 19 de première année sur les graphes, que nous trouverez encore sur la Dropbox de l'année dernière. N'hésitez pas à le consulter de nouveau si besoin.

1.1 - Vocabulaire

Les éléments de vocabulaire ci-dessous doivent être compris et maîtrisés. Voir le chapitre 19 pour les détails.

- Graphe, sommets, **arêtes** (non orientées) ou **arcs** (orientés), notation $G = (S, A)$ ou $G = (V, E)$ (vertices/edges) ;
- Degré d'un sommet (cas non orienté), degrés entrant et sortant d'un sommet (cas orienté) ;
- Lemme des poignées de main :
  - Si G est non orienté, $\sum_{u \in S} \deg(u) = 2|A|$
  - Si G est orienté, $\sum_{u \in S} \deg^+(u) = \sum_{u \in S} \deg^-(u) = |A|$
  - Ces résultats seront très utiles pour analyser la complexité des parcours de graphes, mais aussi potentiellement pour d'autres algorithmes.
- Chemin, distance entre deux sommets, cycle.
- Graphe connexe : toute paire de sommets est reliée par au moins un chemin. Composantes connexes.

1.2 - Représentations informatiques en Python

Un graphe est ce que l'on appelle en informatique une *structure de données abstraite* : on peut décrire précisément un algorithme sur les graphes (comme les algorithmes de parcours) sans expliquer comment les graphes (et leurs opérations) sont gérés en machine.

Les deux représentations de graphe à connaître en CPGE sont les *listes d'adjacence* et les *matrice d'adjacence*. En fonction de la représentation choisie, un même algorithme aura deux implémentations différentes, et donc a priori deux complexités différentes.

Exercice 1. Compléter le tableau de la page suivante, en proposant une implémentation des opérations étudiées et en donnant leurs complexités.

Rappels : listes d'adjacence et matrice d'adjacence

Soit $G = (S, A)$ un graphe (orienté ou non) comportant n sommets, numérotés de 0 à $n-1$.

- La *matrice d'adjacence* de G est la matrice carrée de taille n notée $M = (m_{i,j})_{0 \leq i,j \leq n-1}$ définie par

$$\forall (i,j) \in [\![0\,;n-1]\!]^2 \qquad m_{i,j} = \begin{cases} 1 \text{ si } (i,j) \in A \\ 0 \text{ sinon} \end{cases}$$

On peut représenter une matrice d'adjacence en Python par une liste de listes ou un tableau Numpy 2D.

- Une représentation par *listes d'adjacences* de G est la donnée d'une liste `adj` de taille n, telle que pour tout sommet $i \in [\![0\,;n-1]\!]$, `adj[i]` contient la liste des sommets voisins de i, dans un ordre quelconque.

On utilise alors en Python une liste de listes.

Opération	Matrice d'adjacence	Listes d'adjacences

Récupérer la liste des voisins d'un sommet u (par exemple pour les parcourir ensuite)	`def voisins(u, M):`
	`V = []`
	`for v in range(len(M)):`
	`  if M[u][v] == 1:`
	`  V.append(v)`
	`return V`
	→ $O(\text{len}(M)) = O(\lvert S \rvert)$
		`def voisins(u, adj):`
		`return adj[u]`
		→ $O(\lvert adj[u] \rvert) = O(d(u))$
		Listes meilleures

Savoir si $(u,v) \in A$, étant donnés u et v	`def est_une_arete(u, v, M):`
	`return M[u][v] == 1`
	→ $O(1)$
		`def est_une_arete(u, v, adj):`
		`return v in adj[u]`
		→ $O(\lvert adj[u] \rvert) = O(d(u))$
		Matrices meilleures

Remarque : Dans toute la suite, on ne travaillera qu'avec des graphes représentés par listes d'adjacences, cette structure étant plus efficace pour parcourir les voisins d'un sommet $u \in S$ donné, une opération très utilisée dans les parcours de graphes.

2 - Généralités sur les parcours de graphe

Parcourir un graphe à partir d'un sommet $s \in S$ donné appelé la *source*, c'est explorer les sommets accessibles depuis s en empruntant des arêtes du graphe dans un ordre prédéfini. Le choix de l'ordre d'examen des sommets et des arêtes détermine précisément le type de parcours mis en place. En particulier :

À retenir : détails d'implémentation du parcours en profondeur

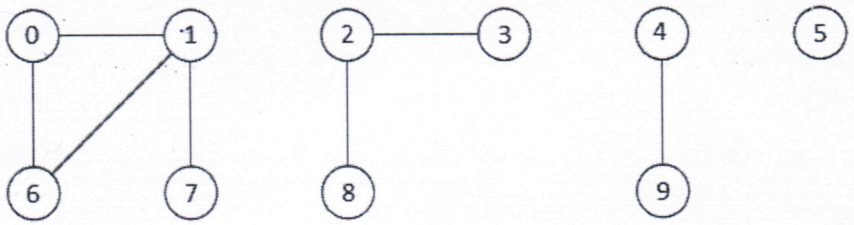
- Il s'agit d'un parcours **naturellement récursif**, dont le pseudo-code est rappelé ci-dessous :

  `Parcours`(u) :
  Pour tout sommet v voisin de u
  Si v n'a pas encore été vu
  Indiquer v comme étant déjà vu, puis lancer `Parcours`(v).

- Pour **retenir quels sommets ont déjà été vus** au courant du parcours, on peut utiliser un **dictionnaire** (dont les clés sont les sommets déjà vus et les valeurs associées arbitraires, comme par exemple dans le DM1) ou **une liste de booléens** (si les sommets sont numérotés à partir de 0).

3.2 - Une application : calcul des composantes connexes d'un graphe non orienté

On souhaite ici adapter l'algorithme de parcours en profondeur afin d'écrire une fonction `composantes(G)` qui prend en argument un graphe non orienté représenté par listes d'adjacences, et renvoie une liste de listes, chaque sous-liste comportant les sommets d'une composante connexe de `G` (dans un ordre quelconque). Sur le graphe représenté ci-dessous, on veut par exemple renvoyer `[[0,1,6,7], [2,3,8], [4,9], [5]]`.



Exercice 3. On propose ci-dessous des choix d'implémentations, ainsi que différentes étapes préliminaires à la conception d'un tel programme. Compléter les pointillés, en consultant si besoin le corrigé (bientôt) sur la Dropbox.

- On traduira en Python le pseudo-code du parcours en profondeur **à l'intérieur** de la fonction `composantes`.
- On utilisera une liste de booléens `dejaVu`. Comment l'initialiser ? `n = len(G)` `dejaVu = [False]*n` `#n = nbre de sommets`
- Quelle(s) information(s) relative(s) aux composantes connexes le parcours en profondeur à partir du sommet 0 permet-il d'obtenir ? Le parcours de G à partir de 0 permet d'obtenir les sommets de la composante connexe contenant 0.
- Proposer alors une méthode **simple** pour obtenir toutes les composantes connexes, chacune une seule fois.

Pour obtenir toutes les composantes, on peut alors utiliser une boucle pour parcourir tous les sommets dans l'ordre et lancer un parcours pour chaque sommet qui n'a pas encore été vu auparavant (sinon, il se trouverait déjà dans une composante).

- Pour gérer les composantes connexes, on utilise deux listes : `comp` contient les sommets de la composante en cours de construction, tandis que `compos` contient la liste des composantes connexes déjà construites précédemment.

Exercice 4. À l'aide du travail préliminaire ci-dessus, écrire la fonction `composantes(G)`. N'oubliez pas de la programmer sur machine et de la tester, par exemple sur le graphe ci-dessus.

Au programme au prochain cours...

Analyse de complexité des parcours, parcours en largeur (rappels) et Algorithme de Dijkstra (nouveau).