Algorithmique des graphes

1 Représentations d'un graphe

On rappelle quelques définitions :

Définition 1.
- *Un graphe est donné par un ensemble de n sommets (numérotés usuellement, en informatique, de 0 à $n-1$) ; et un ensemble d'arêtes qui relient ces sommets.*
- *En version non orientée, une arête relie deux sommets i et j : on la représente alors par l'*ensemble $\{i, j\}$.
- *En version orientée, une arête va d'un sommet i à un sommet j. L'ordre importe donc ; on représente une arête par le* couple (i, j)
- *On dit que 2 sommets i et j sont voisins s'il existe une arête $\{i, j\}$ (cas non orienté) ou une arête (i, j) (cas orienté). Ainsi, dans le cas non-orienté :*

$$i \text{ est voisin de } j \Leftrightarrow j \text{ est voisin de } i$$

alors qu'en orienté ce n'est pas forcément le cas.

Pour représenter un graphe à n sommets en Python il existe principalement deux structures de données :

- La matrice d'adjacence : elle appartient à $\mathcal{M}_n(\mathbb{R})$, et son coefficient vaut 1 ssi les sommets i et j sont voisins.
  On voit que dans un graphe non orienté la matrice d'adjacence est symétrique ; ce n'est pas forcément le cas si le graphe est orienté.
- Les listes d'adjacence : le graphe est représenté par une liste de n listes ; la liste d'indice i contient alors l'ensemble des sommets voisins du sommet i.

Chaque structure a son intérêt ; il faut savoir passer de l'une à l'autre.

Exercice 1.

1. Coder une fonction `mat_vers_listes` qui prend en argument la matrice d'adjacence (codée par un `np.array`) d'un certain graphe ; et renvoie les listes d'adjacence de ce graphe.
2. Coder une fonction `listes_vers_mat` qui prend en argument les listes d'adjacence d'un certain graphe ; et renvoie la matrice d'adjacence de ce graphe.

Munis de ces structures nous allons nous intéresser à deux problèmes algorithmiques sur les graphes (mais il y en a plein d'autres !) : la recherche de composantes connexes, et le tri topologique.

2 Connexité

Un graphe est dit connexe si on peut relier deux sommets quelconques de ce graphe peuvent être reliés par une chaîne (cas non orienté) ou un chemin (cas orienté).

Si i est un sommet du graphe, on appelle *composante connexe de i* l'ensemble des sommets reliés à i par une chaîne (ou un chemin). Si un graphe est connexe, la composante connexe de tout sommet est le graphe entier ; sinon il existe plusieurs composantes connexes disjointes.
Un problème algorithmique est donc de savoir si un graphe donné est connexe ; et, s'il ne l'est pas, de déterminer ses composantes connexes.
Nous allons donner deux manières de procéder : une manière « brutale » ; et une manière plus fine qui nous permettra d'introduire les méthodes de *parcours* d'un graphe.

2.1 Algo brutal

On propose l'algorithme suivant :

- initialement on étiquette les sommets de la manière suivante : le sommet i a l'étiquette i.
- on effectue ensuite l'opération suivante : on parcourt toutes les arêtes du graphe ; et quand on rencontre deux sommets voisins i et j on leur donne l'étiquette $\min(i, j)$. Ainsi ces deux sommets, reliés par une arête, ont maintenant la même étiquette.
- on répète cette opération tant qu'un parcours des arêtes mène encore à une modification des étiquettes.

À l'issue de cet algo, deux sommets reliés par un chemin auront forcément la même étiquette : sinon en empruntant un chemin qui va de l'un à l'autre on parcourt à un moment une arête qui relie deux sommets d'étiquette différente, ce qui est absurde car l'algo a terminé.
Et deux sommets qui ne sont pas reliés par un chemin auront des étiquettes différentes. Considérons en effet deux sommets i et j avec $i \neq j$. S'ils se retrouvent avec la même étiquette k à la fin de l'algo, ils ont « hérité » cette étiquette du sommet k ; les étiquettes se propageant le long des arêtes il ya une chaîne reliant i et k et une chaîne reliant j et k. On en déduit une chaîne reliant i et j.
Par contraposée, s'il n'y a pas de chaîne reliant i et j ils n'auront pas la même étiquette.

Mettons en œuvre cet algorithme sur un graphe représenté par sa matrice d'adjacence.

- Si `M` est la matrice d'adjacence du graphe, `len(M)` est son nombre de lignes ; ce qui permet de récupérer le nombre n de sommets. On numérote les sommets de 0 à $n-1$.
- On initialise une liste d'étiquettes `T` : initialement l'étiquette de chaque sommet est égale à son numéro.
- Le parcours des arêtes du graphe se fera en parcourant toute la matrice d'adjacence du graphe (avec 2 boucles `for`) ; si on rencontre un coefficient `M[i,j]` égal à 1, et `T[i]` est différent de `T[j]`, on donne à `T[i]` et `T[j]` la valeur `min(T[i],T[j])`.
- La partie la moins intuitive est la gestion de la condition de poursuite de l'algorithme : « tant qu'une modification a été effectuée, on continue » .
  Pour faire cela on pourra maintenir un booléen `modif` et introduire une boucle `while modif == True:` (en fait élégamment raccourci en `while modif:`).
  Au début `modif` vaudra `True` pour qu'on démarre la boucle ! Ensuite à chaque tour de boucle on commence par donner à `modif` la valeur `False` ; et on le remet à `True` dès qu'une modification est effectuée.

Exercice 2. Coder cet algorithme.

2.2 Algo élégant : parcours de graphe

De manière générale, *parcourir* un graphe consiste à :

- considérer un sommet du graphe, qu'on marque comme « visité »
- étant sur un sommet visité, aller visiter ses voisins (en se promenant sur les arêtes du graphe).

Il existe deux types de parcours :

1. Le parcours en largeur (BFS : Breadth-First Search) : une fois qu'un sommet est visité, on visite tous ses voisins immédiats[1] ;
2. Le parcours en profondeur (DFS : Depth-First Search) : une fois qu'un sommet est visité, on visite un de ses voisins, puis un voisin de ce voisin, etc.

Ces deux algorithmes permettent de recenser tous les sommets « accessibles » depuis le sommet de départ : c'est-à-dire sa composante connexe. Les sommets sont découverts dans un ordre différent, et suivant les problèmes posés un algorithme peut être plus efficace que l'autre. Par exemple, BFS est meilleur pour calculer le plus court chemin entre deux sommets ; alors que DFS sera plus indiqué pour sortir d'un labyrinthe.

On va déterminer les composantes connexes d'un graphe donné à l'aide d'un parcours en largeur.
On considère un graphe à n sommets, non orienté, modélisé par ses listes d'adjacence (pour changer). On définit le parcours en largeur d'un graphe par l'algorithme suivant.

On crée une liste `visites` qui associe à chaque sommet un statut : la composante de cette liste est nulle si le sommet n'est pas visité, et vaut 1 s'il est visité.

On crée une liste `a_explorer`, initialement vide, qui contiendra les sommets visités, à partir desquels il faut continuer l'exploration.

On choisit un sommet i non visité du graphe ; on le marque comme visité et on l'ajoute à la liste `a_explorer`.

Tant que la liste `a_explorer` n'est pas vide :

- on retire son premier élément ;
- si des voisins de ce sommet n'ont pas encore été visités, on les marque visités et on les ajoute à la fin de la liste `a_explorer`

À l'issue de ces opérations, les sommets marqués comme visités forment exactement la composante connexe de i.

Ensuite on repart d'un sommet non encore visité (s'il y en a un), et on récupère de même sa composante connexe. Quand tous les sommets ont été visités, on a ainsi toutes les composantes connexes.

[1] C'est celui sur lequel est basé le problème d'algorithmique d'ECRICOME 2023.

2.2.1 Composante connexe d'un sommet donné

Compléter la fonction suivante qui prend pour arguments les listes d'adjacence d'un graphe à n sommets, et un sommet $i \in [\![0, n-1]\!]$, et renvoie la liste des sommets formant la composante connexe de i.

```
def comp_conn(L,i):
    """ composante connexe du sommet i, du graphe avec listes d'ajacence L"""
    n = len(L) ## nb de sommets
    visites = ................. ## statut d'un sommet (visité ou pas)
                                ## initialement aucun sommet n'est visité
    ................. ## on marque i comme visité
    a_explorer = .................  ## et on l'ajoute à la liste "à explorer"
    while len(a_explorer)>0:
        j=a_explorer.pop(0) ## commande HP : retire le premier sommet de la liste
                            ## a_explorer, et stocke sa valeur dans une variable j
        ### ici on déclare visités les voisins de j qui ne l'étaient pas encore
        ### et on les ajoute à la liste a_explorer
        ......
        ......
        ......
        ......
    return [i for i in range(n) if visites[i]==1] ## on renvoie la liste
                                                  ## des sommets visités
```

2.2.2 Obtention de toutes les composantes connexes

Enfin il faut une dernière étape : retirer les sommets de cette composante connexe du graphe, et lancer une nouvelle recherche de composante connexe sur un des sommets restants... ceci tant qu'il reste des sommets non explorés.
On part donc de la liste des sommets

```
somm = list(range(len(L)))
```

et on initialise une liste de composante connexes (initialement vide, donc)

```
compo=[]
```

Tant que la liste des sommets n'est pas vide, on calcule la composante connexe du premier sommet de la liste ; puis on retire tous les sommets obtenus de la liste `somm` (avec la méthode `somm.remove(...)`).

Coder cet algorithme.

3 Le tri topologique

3.1 Algorithme de Kahn

On se place ici dans le cas d'un graphe orienté sans boucles, et sans cycle. On appelle *tri topologique* des sommets du graphe une numérotation des sommets telles que toute arête va d'un sommet i à un sommet j, avec $j > i$.
On voit que si le graphe admet un cycle, un tel tri n'existe pas ; l'algorithme que nous allons coder montre que si ce n'est pas le cas, un tri topologique existe.

Nous allons coder l'*algorithme de Kahn.* Le principe est le suivant.

- On initialise une liste vide.
- Dans le graphe, on cherche un sommet de degré entrant 0 (c'est-à-dire, il n'existe aucune arête qui arrive à ce sommet).
  Un tel sommet existe si le graphe est acyclique. En effet, si tout sommet était le point d'arrivée d'une arête : considérons un sommet quelconque et déplaçons-nous vers un de ses prédécesseurs. Par hypothèse on peut répéter cette opération à l'infini ; comme le nombre de sommets du graphe est fini on va repasser 2 fois par le même sommet, formant ainsi un cycle. Contradiction.
- On retire ce sommet du graphe (*ie :* on retire le sommet, et toutes les arêtes qui partent de lui ou qui y arrivent), et on l'ajoute à la liste.
- Dans le graphe restant, il existe encore un sommet de degré entrant 0 (sinon ce sous-graphe comporterait aussi un cycle, qui serait alors un cycle dans le graphe d'origine : contradiction) : on retire ce sommet et on l'ajoute à la fin de la liste.
- ...etc...
- à la fin il ne reste qu'un seul sommet et plus d'arêtes : ce sommet est ajouté en dernière position de la liste et l'algo est terminé.

On justifie ici que cet algo fournit un tri topologique (ce qui, au passage, montre l'existence d'un tel tri).
Soit une arête $u \to v$: il s'agit de montrer que le numéro assigné à u est strictement inférieur à celui assigné à v.
Au cours de l'algorithme, à l'étape où v reçoit son numéro il est de degré entrant 0, ce qui implique que l'arête $u \to v$ a été retirée du graphe, et donc le sommet u a déjà été retiré. Ceci montre que le numéro de u est strictement inférieur à celui de v.

3.2 À quoi ça sert ?

On peut citer plusieurs applications :

- on veut effectuer une liste de tâches qui dépendent les unes des autres (ie : la tâche i ne peut être entreprise que si la tâche j a été effectuée – par exemple pour faire le toit d'une maison il faut avoir monté les murs). On représente cette situation par un graphe dont les sommets sont les tâches à accomplir, et une arête va de la tâche i à la tâche j ssi la tâche i doit être effectuée avant la tâche j. Le tri topologique de ce graphe donne un ordre dans lequel accomplir les tâches pour satisfaire les dépendances.
- En informatique, l'installation d'un logiciel peut demander, au préalable, que d'autres soient installés. Le tri topologique donnera un ordre dans lequel effectuer les installations.

3.3 Mise en œuvre

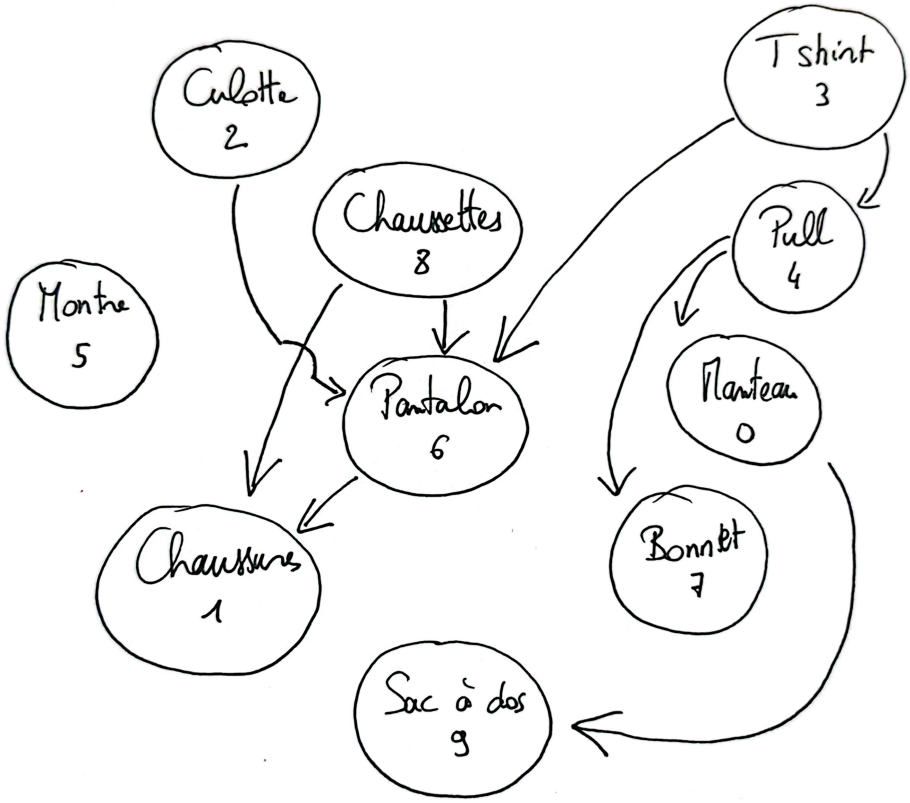
La petite Blanche C., 5 ans, s'habille seule tous les matins ; mais elle s'aperçoit qu'elle ne peut pas mettre ses vêtements dans un ordre quelconque : par exemple elle doit mettre son pantalon avant ses chaussures. Nous allons l'aider à trouver un ordre dans lequel passer ses vêtements.
On considère la liste de vêtements suivante :

0	1	2	3	4	5	6	7	8	9
manteau	chaussures	culotte	t-shirt	pull	montre	pantalon	bonnet	chaussettes	sac à dos

implémentée en Python par

```
vetements=['manteau','chaussures','culotte',
           't-shirt','pull','montre','pantalon',
           'bonnet','chaussettes','sac à dos']
```

On représente ci-dessous un graphe dont les 10 sommets représentent les 10 vêtements, et où une arête va de i à j ssi le vêtement i doit être passé avant le vêtement j.



(oui, croyez-moi, il est bien plus pratique de mettre ses chaussettes et son t-shirt avant son pantalon quand on a 5 ans)

On voit alors qu'un tri topologique des sommets de ce graphe donne un ordre possible dans lequel mettre ses vêtements.
Dans le reste du TP il sera commode de numéroter les sommets de 0 à 9, mais la liste précédente nous permettra de remonter aux vêtements auxquels ils font référence.
On va ici utiliser la représentation du graphe par matrice d'adjacence pour implémenter cet algorithme (exo : le refaire avec les listes).

Exercice 3. Soit $M \in \mathcal{M}_n(\mathbb{R})$ la matrice d'adjacence d'un graphe orienté acyclique, dont les sommets sont numérotés de 0 à $n-1$. Comment repérer, sur la matrice, un sommet de degré entrant 0 ?
Proposer une fonction `deg_entrant_nul` qui prend en argument M et renvoie le numéro d'un sommet de degré entrant nul (ici on considère les sommets sont numérotés comme les indices de la matrice d'adjacence).

Exercice 4. On veut maintenant « enlever un sommet » au graphe. À quelle opération sur la matrice d'adjacence cela correspond-il ?
Compléter la fonction suivante qui prend pour argument une matrice M et deux entiers i et j et renvoie la matrice obtenue en retirant à M sa i-ème ligne et sa j-ème colonne.

```
def enlever_lignecol(M,i,j):
    """ prend une matrice M et renvoie cette matrice privée de la
    i-ème ligne et j-ème colonne"""
    (n,p) = np.shape(M) # format de la matrice
    l = ..............................  # la liste [0,1,...,i-1,i+1,...,n-1]
    c = ..............................  # la liste [0,1,...,j-1,j+1,...,p-1]
    return np.array([[M[i,j] for j in ....] for i in ...])
```

Piège : si on enlève un sommet au graphe, il ne faut pas que les numéros des sommets changent !
Par exemple, si on enlève le sommet 3 à un graphe de sommets 0,1,2,3,4, le graphe restant est représenté par une matrice d'adjacence de $\mathscr{M}_4(\mathbb{R})$. Mais les sommets de ce nouveau graphe doivent être numérotés 0,1,2,4 et non 0,1,2,3 comme ils le seraient « par défaut » !
Dans notre code il faudra donc maintenir à jour la liste des numéros des sommets dans une variable `liste_sommets` initialement égale à `[0,1,2,...,n]` ; à laquelle on enlèvera les sommets au fur et à mesure qu'ils sont ajoutés à l'ordre topologique.

On rappelle pour cela que si L est une liste, la commande

```
del L[i]
```

enlève à la liste son élément d'indice `i`.

Nous sommes maintenant prêts à coder le tri topologique ! Compléter la fonction suivante à cet effet.

```
def tri_topo(M):
    """ effectue un tri topologique des sommets
    d'un graphe orienté acyclique de matrice
    d'adjacence M"""
    tri=[] # contiendra la liste des sommets triés
    n=np.shape(M)[0] # le nombre de sommets du graphe
    liste_sommets = ..... # contient initialement tous les sommets du graphe
    for k in range(n):
        somm = ..............  # l'indice d'un sommet de degré entrant nul
        ....................  # qu'on ajoute à la liste des sommets triés
        ....................  # on le retire ensuite du graphe
        ....................  # et de la liste de ses sommets

    return tri
```

Si on passe en argument la matrice d'adjacence des vêtements de Blanche, la fonction fournira en sortie une liste contenant les entiers de 0 à 9 dans un certain ordre. Blanche est encore petite, et aimerait bien que l'ordinateur lui fournisse une liste de vêtements.

Exercice 5. Programmer une fonction qui prend en argument la liste renvoyée par la fonction `tri_topo`, et renvoie la liste des vêtements correspondants. Autrement dit, il faut renvoyer la liste des `vetements[i]` où `i` parcourt la liste des sommets ordonnés par l'ordre topologique.

3.4 Une variante inutile

Modifier les fonctions précédentes pour que, à chaque étape de l'algorithme de Kahn, on retire un sommet choisi de manière aléatoire parmi ceux de degré entrant nul. Blanche aura ainsi plein d'ordres possibles dans lequel enfiler ses vêtements !