Devoir ITC n°4

Réseau social

Ce sujet aborde plusieurs questions concernant les réseaux sociaux en ligne, modélisés comme des graphes orientés. Dans tout le sujet, on nomme **réseau social** un graphe orienté $G = (S, A)$ avec S un ensemble de sommets (les individus) et A un ensemble de couples $(u, v) \in S \times S \setminus \{u\}$ appelés arcs ; la présence de l'arc (u, v) représente le fait que u « suit » v. On note n_S le nombre de sommets et n_A le nombre d'arcs. Notons que la définition proposée des arcs exclut les boucles, c'est-à-dire les couples (u, u). Un exemple de tel réseau est représenté en figure 1.

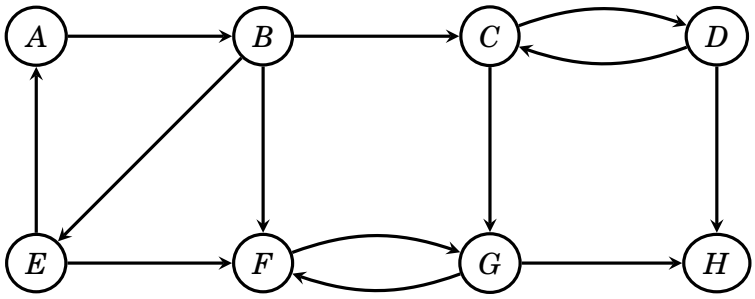


Figure 1 : Exemple de réseau

Dans tout le sujet, les questions liées à la complexité attendent une réponse en fonction de n_S et/ou n_A, et devront être justifiées succintement.

Dans tout le sujet, on représente le graphe par un dictionnaire de dictionnaire d'adjacence, c'est-à-dire que si g est un dictionnaire représentant le graphe, alors ses clés sont les noms des sommets du graphe ; et si u est un sommet du graphe, alors g[u] est un dictionnaire dont les clés sont les sommets v tels que (u, v) soit un arc du graphe (la valeur associée est prise égale à 1). Ainsi, une définition du graphe de la figure 1 commencerait par

```
g = {'A': {'B': 1}, 'B': {'C': 1, 'E': 1, 'F': 1}, 'C': {'D: 1', 'G': 1}, ... }
```

Une telle représentation sera nommée **dictionnaire d'adjacence**. Par défaut, c'est celle que nous utiliserons dans le sujet.

On peut également représenter un graphe par sa **matrice d'adjacence** : si on étiquette les sommets par des entiers 0, 1, 2, ... $n_s - 1$, soient deux sommets u, v représentés par les indices i, j, alors

$$M_{ij} = \begin{cases} 1 & \text{si } (u, v) \in A \\ 0 & \text{sinon} \end{cases}$$

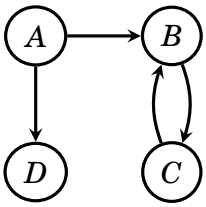
I. Généralités

Soit le graphe dont le dictionnaire d'adjacence est

```
g1 = {'A': {'B': 1, 'D': 1}, 'B': {'C': 1}, 'C': {'B': 1}, 'D': {}}
```

1 – Dessiner ce graphe et donner sa matrice d'adjacence.

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



2 – Écrire une fonction `dict_to_mat(g: dict) -> list` qui, étant donné un graphe donné sous forme de dictionnaire d'adjacence, renvoie une matrice (liste de listes) de taille n_S. On suppose pour simplifier que les sommets sont parcourus dans l'ordre de leurs indices.

```
def dict_to_mat(g):
    mat = []
    for u in g.keys():
        ligne = []
        for v in g.keys():
            if v in g[u]:
                ligne.append(1)
            else:
                ligne.append(0)
        mat.append(ligne)
    return mat
```

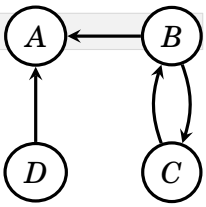
On définit le graphe transposé $G^T = (S, A^T)$ comme le graphe contenant les mêmes sommets que G, mais dont les arcs sont inversés :

$$(u, v) \in A \quad \Rightarrow \quad (v, u) \in A^T$$

Dans le cadre d'un réseau social, le graphe transposé représente la direction dans laquelle l'information se propage.

3 – Dessiner le graphe transposé de g1 et donner sa représentation en dictionnaires d'adjacence.

```
g1_t = {'A': {}, 'B': {'A': 1, 'C': 1}, 'C': {'B': 1}, 'D': {'A': 1}}
```



4 – Écrire une fonction `transpose(g: dict) -> dict` qui renvoie le graphe transposé de g, sans modifier g. La fonction devra être en $\mathcal{O}(n_S + n_A)$ (on ne le justifiera pas).

```
def transpose(g):
    tr = {u: {} for u in g.keys()}
    for u in g.keys():
        for v in g[u]:
            tr[v][u] = 1
    return tr
```

Une caractérisation simple de la centralité d'un individu dans le réseau est le nombre d'arc qui arrivent sur cet individu, appelé degré entrant.

5 – Quel est le sommet le plus central dans le réseau de la figure 1 ? Écrire une fonction `sommet_central(g: dict) -> str` qui renvoie un sommet dont le degré entrant est maximal (en cas d'égalité, on en renverra un). Préciser la complexité.

Le sommet F.

```
def sommet_central(g):
    # on calcule tous les degrés entrants : O(nS + nA)
    degrés = {u: 0 for u in g}
    for u in g:
        for v in g[u]:
            degrés[v] += 1
    # on trouve le maximum : O(nS)
    deg_max, sommet_max = 0, ''
    for u in degrés:
        if degrés[u] > deg_max:
            deg_max, sommet_max = degrés[u], u
    return sommet_max
```

Le parcours pour calculer le degré entrant parcourt une fois tous les sommets ($\mathcal{O}(n_S)$) et une fois chaque arc partant de chaque sommet, donc une fois tous les arcs ($\mathcal{O}(n_A)$), soit une complexité $\mathcal{O}(n_S + n_A)$. La deuxième boucle parcourt les sommets une fois chacun, donc $\mathcal{O}(n_S)$, et la complexité totale est donc $\mathcal{O}(n_S + n_A)$.

Autre proposition : les degrés sortants sont plus faciles à compter, on peut donc exploiter `transpose`

```
def sommet_central(g):
    g_T = transpose(g)
    maxi, u_max = 0, ''
    for u, adj_u in g_T.items():
        if lent(adj_u) > maxi:
            maxi, u_max = len(adj_u), u
    return u_max
```

La boucle n'est que en $\mathcal{O}(n_S)$, mais la fonction `transpose` est en $\mathcal{O}(n_S + n_A)$, on retrouve donc finalement la même complexité.

II. Cliques et célébrités

Une **clique** est un ensemble d'individus qui se connaissent tous mutuellement ; formellement, une partie S' de S est une clique si et seulement si

$$\forall (u,v) \in S' \times S' \setminus \{u\}, \quad (u,v) \in A$$

6 – Déterminer les deux cliques du graphe de la figure 1.

C-D et F-G.

---

7 – Écrire une fonction `est_clique(g: dict, sommets: list) -> bool` qui prend en paramètres un graphe et une liste de sommets, et ranvoie `True` si la liste est une clique, et `False` sinon.

```
def est_clique(g, sommets):
    n = len(sommets)
    for u in sommets:
        for v in sommets:
            if u != v and (u not in g[v] or v not in g[u]):  # test in en O(1)
                return False
    return True
```

Pour chercher une cliques, on propose l'algorithme suivant :

```
def clique_possible(g):
    cl = []
    for u in g.keys():
        if len(cl) == 0:
            cl = [u]
            v = u
        elif u in g[v] and v in g[u]:
            cl.append(u)
        elif len(cl) == 1:
            cl = [u]
            v = u
    return cl
```

8 – Décrire l'évolution des variables `u` et `cl` à chaque tour de boucle dans le cas du traitement du graphe de la figure 1 (on supposera que les sommets sont parcourus dans l'ordre alphabétique). À quelle famille d'algorithmes appartient-il ?

- u = 'A', cl =['A']
- u = 'B', cl =['B']
- u = 'C', cl =['C']
- u = 'D', cl =['C', 'D']
- u = 'E', cl =['C', 'D']
- u = 'F', cl =['C', 'D']
- u = 'G', cl =['C', 'D']
- u = 'H', cl =['C', 'D']

Il s'agit d'un algorithme **glouton**.

---

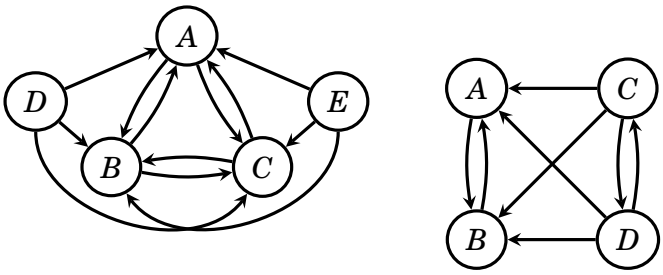
9 – Donner la complexité de cet algorithme.

On parcourt simplement les sommets une fois, donc $\mathcal{O}(n_S)$.

On définit à présent une **clique de célébrités** C comme une clique dans laquelle tous les individus sont connus de tout le monde, mais ne connaissent que les autres célébrités :

$$\forall (c,u) \in C \times S, \quad ((u,c) \in A) \ \wedge \ ((c,u) \in A \Rightarrow u \in C)$$

10 – Dessiner un graphe contenant 5 individus et une clique de célébrités de taille 3, puis un graphe contenant 4 individus, une clique de célébrités de taille 2 et une autre clique.



11 – Soit $G = (S, A)$ un graphe quelconque. Montrer que s'il existe une clique de célébrités non vide C dans G, alors celle-ci est unique.

Supposons qu'il existe deux cliques de célébrités non vides distinctes C_G et C'_G. Soient $c \in C_G$ et $c' \in C'_G$. D'après la définition d'une clique de célébrités,

$$c' \in C'_G \quad \Rightarrow \quad (c, c') \in A$$

or c est une célébrité de C_G donc

$$(c, c') \in A \quad \Rightarrow \quad c' \in C_G$$

ce qui contredit l'hypothèse de C_G et C'_G distincts. On en déduit l'unicité de la clique de célébrités.

12 – Écrire une fonction `célébrités(g: dict, clique: list) -> bool` qui prend en paramètres un graphe et une clique du graphe, et renvoie `True` si celle-ci est une clique de célébrités et `False` sinon.

```
def célébrités(g, clique):
    for u in g:
        for v in clique:
            if v != u and v not in g[u]:
                return False
    return True
```

III. Composantes fortement connexes

Les algorithmes de recherche de cliques sont plus complexes, mais commencent systématiquement par une décomposition en composantes fortement connexes du graphe, puisqu'une clique est nécessairement entièrement contenue dans une telle composante. On rappelle qu'une composante fortement connexe $C \subset S$ est telle que $\forall (u,v) \in C \times C$, il existe un chemin de u à v.

L'algorithme classique de partition s'appuie sur un parcours en profondeur, au cours duquel il faut noter les dates de découvertes d'un sommet ainsi que sa date de fin d'exploration, la date étant un entier croissant à chaque découverte d'un nouveau sommet. Par exemple, on représente ci-dessous le résultat de l'exploration du graphe de la figure 1, avec sur chaque nœud sa date de découverte et sa date de fin d'exploration :

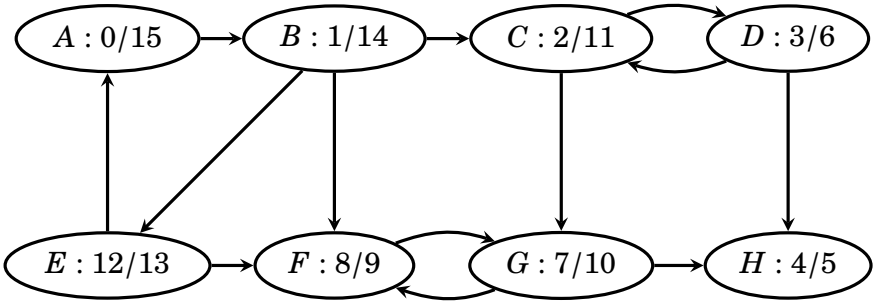


Figure 2 : Parcours en profondeur

On rappelle la méthode du parcours en profondeur : chaque sommet possède un état (`'vert'`, `'bleu'` ou `'rouge'`), une date de découverte et une date de fin. On **visite** un sommet u de la façon suivante :

- on le marque comme découvert : son état passe de `'vert'` à `'bleu'` ;
- sa date de début de traitement est notée comme la date courante ;
- pour chaque sommet v dans l'état `'vert'` de son dictionnaire d'adjacence, on augmente la date courante de 1 et on visite récursivement le sommet v ;
- la date courante devient alors la date de fin de traitement de v ;
- une fois tous les sommets adjacents traités, on incrémente la date courante, on note la date de fin de traitement de u, et son état devient `'rouge'`.

Le parcours consiste alors à effectuer des visites sur tous les sommets verts du graphe.

On stocke l'état et les dates de début et de fin de traitement dans des dictionnaires dont les clés sont les noms des sommets ; par exemple, `état[u] = 'bleu'` enregistre l'état bleu pour le sommet u, `date_d[u] = date` enregistre la valeur `date` dans l date de début de traitement du sommet u, et `date_f[u]` contient la date de fin du traitement du sommet u. Le dictionnaire des états est initialisé avec la valeur `'vert'` pour tous les sommets.

13 – Écrire une procédure **récursive**

```
visite_pp(g: dict, u: str, état: dict, date_courante: int, date_d: dict, date_f: dict) -> None
```

qui visite le sommet u du graphe g et mettra à jour les états et les dates de début et de fin de traitement.

```
def visite_pp(g, u, état, date_courante, date_d, date_f):
    état[u] = 'bleu'
    date_d[u] = date_courante
    for v in g[u]:
        if état[v] == 'vert':
            date_courante += 1
            visite_pp(g, v, état, date_courante, date_d, date_f)
            date_courante = date_f[v]
    état[u] = 'rouge'
    date_f[u] = date_courante + 1
```

---

14 – En déduire une fonction `parcours_p(g: dict) -> dict` qui effectue un parcours en profondeur de l'ensemble du graphe et renvoie le dictionnaire des dates de fin de traitement. Quelle est sa complexité ?

```
def parcours_p(g):
    # initialisation
    état = {u: 'vert' for u in g.keys()}
```

```
    date_d = {u: 0 for u in g.keys()}
    date_f = {u: 0 for u in g.keys()}
    date = 0
    # parcours
    for u in g.keys():
        if état[u] == 'vert':
            visite_pp(g, u, état, date, date_d, date_f)
            date = date_f[u] + 1
    return date_f
```

Le système de marquage des sommets permet de s'assurer que chaque sommet sera visité une et une seule fois, tandis que la boucle dans `visite_pp` teste une seule chaque arc partant d'un sommet u donné. On en déduit que l'algorithme visite une et une seule fois chaque sommet et chaque arc, donc la complexité est $\mathcal{O}(n_A + n_S)$.

Pour finir la décomposition en composantes fortement connexes, on effectue un nouveau parcours en profondeur, sur le graphe transposé, en partant des sommets dans l'ordre inverse de date de fin du premier parcours. Chaque ensemble de sommets accessibles depuis un même sommet de départ se trouve alors dans une même composante fortement connexe, que l'on peut remplir au fur et à mesure où on découvre les sommets.

15 – Expliquer succintement quelles modifications de `parcours_p` et `visite_pp` permettent d'effectuer cette deuxième étape, en supposant que l'on dispose de la liste des sommets triés en ordre de date de fin décroissante.

- dans l'initialisation, il faut créer une liste pour les composantes connexes et calculer le graphe transposé que l'on visitera ;
- avant d'amorcer une nouvelle visite, on créé une liste pour représenter la composante en cours de construction ;
- cette liste doit être passée à `visite_pp`, qui y ajoute u ;
- on rassemble chaque composasnte dans la grande liste du début ;
- (accessoire) on peut entièrement enlever la gestion des dates de parcours.

On s'intéresse finalement au fait de générer la liste des sommets triée par ordre décroissant de date de fin de traitement.

16 – Écrire une fonction `tri(sommets: list, dates: dict) -> list` qui renvoie une copie triée des sommets triés par date de fin décroissante, fournies dans le dictionnaire `dates`. La complexité (moyenne et/ou dans le pire cas) doit être $\mathcal{O}(n_S \log n_S)$ (on ne la justifiera pas).

On propose un tri par partition-fusion.

```
def tri(tab, date):
    n = len(tab)
    if n < 2:  # cas de base
        return tab
    # appel récursif
    t1 = tri(tab[:n//2], date)
    t2 = tri(tab[n//2:], date)
    # fusion
    tf = [0]*n
    t1.append('sentinelle')  # sentinelle
    t2.append('sentinelle')  # sentinelle
    date['sentinelle'] = -1
    i1, i2 = 0, 0
    for i in range(n):
```

```
        if date[t2[i2]] > date[t1[i1]]:
            tf[i] = t2[i2]
            i2 += 1
        else:
            tf[i] = t1[i1]
            i1 += 1
    return tf
```

On aurait aussi pu faire un tri pivot :

```
def tri2(tab, date):
    n = len(tab)
    if n < 2:
        return tab
    # partition
    t1, t2, piv = [], [], tab[n-1]
    for i in range(n-1):
        if date[tab[i]] > date[piv]:
            t1.append(tab[i])
        else:
            t2.append(tab[i])
    # appel récursif
    t1 = tri(t1, date)
    t2 = tri(t2, date)
    return t1 + [tab[i]] + t2
```