Devoir ITC n°4

Réseau social

Le sujet suivant se compose de 3 parties indépendantes. Vous pouvez traiter les différentes parties dans l'ordre que vous souhaitez.

Lorsqu'une fonction est demandée par l'énoncé, même si vous ne proposez pas de code pour la définir, **vous pouvez supposer qu'elle est définie et vous en servir dans les questions suivantes**.

Dans tout le sujet, les fonctions sont données avec leur signature sous forme d'annotations : par exemple, `fonction(x:int, y:float) -> list` signifie que la fonction prend deux paramètres x (de type entier) et y (de type flottant) et renvoie une liste. Dans le sujet, nous définissons le type `Mat` comme étant une matrice, c'est-à-dire une liste de listes comme défini au-dessus. **Il n'est pas demandé d'écrire les annotations dans la copie**.

Les points suivants seront pris en compte à la notation :

- vous attacherez la plus grande importance à la clarté, à la précision et à la concision de la rédaction ; 1 point pourra être attribué en bonus ou en malus selon la lisibilité de la copie ;
- de même, il est demandée de **tirer un trait entre deux questions** afin de les délimiter clairement ;
- la syntaxe sera évaluée de façon bienveillante, des erreurs mineures ne seront pas sanctionnées ; cependant, une syntaxe Python complètement fantaisiste fera perdre des points de présentation ;
- pour chaque fonction, un point sera attribué si celle-ci a une approche naturelle pour répondre au problème ;
- il est fortement recommandé de délimiter les niveaux d'indentation avec des traits verticaux ;
- si vous êtes amené à repérer ce qui vous semble être une erreur d'énoncé, vous le signalerez sur votre copie et devrez poursuivre votre composition en expliquant les raisons des initiatives que vous avez prises ;
- il est essentiel de respecter le temps imparti : les stylos seront levés à la fin de la composition ;
- le sujet ne sera **pas** rendu avec la copie.

Sur ce, bon courage ☺

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.

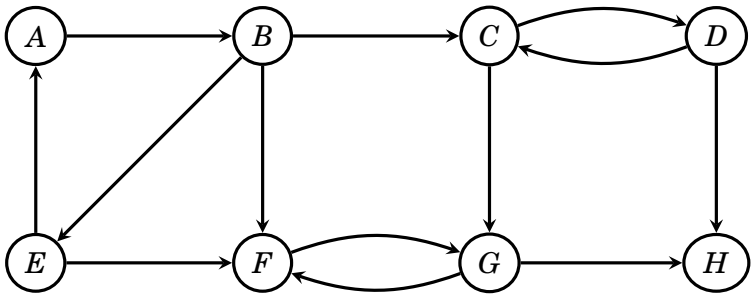

Figure 1 : Exemple de réseau

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 $0, 1, 2, \dots n_s - 1$, 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.

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.

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.

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).

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) ->` renvoie un sommet dont le degré entrant est maximal (en cas d'égalité, on en renverra un). Préciser la 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.

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.
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 ?

9 – Donner la complexité de cet algorithme.
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.

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.

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 :

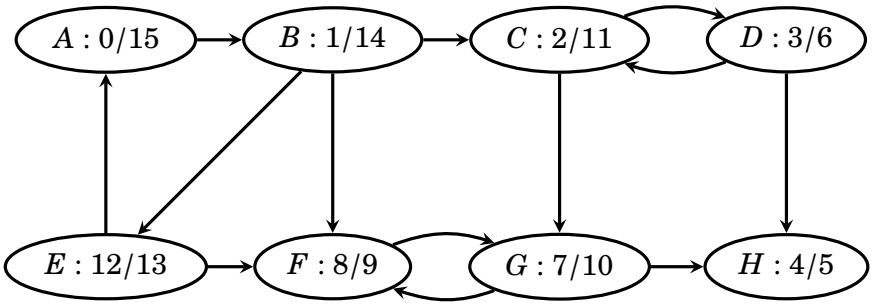

Figure 2 : Parcours en profondeur

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.

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 ?
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.
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).