DS 5 d'informatique de l'année 2022/2023 - MPSI

Calculatrices interdites.

Pensez à numéroter vos feuilles.

Sur votre copie, les questions doivent apparaître dans l'ordre du sujet.

Si vous repérez une erreur d'énoncé, signalez le sur votre copie et poursuivez votre composition.

Tous les graphes considérés sont orientés. Si besoin, on pourra utiliser les files à double extrémité du module `collections` dont voici un exemple d'utilisation :

```
import collections
### Création d'une file à double extrémités (FDE)
L = collections.deque([5,7,3]) # L vaut [5,7,3]
print(len(L))                  # Affiche 3
### Ajout au début de la FDE
L.appendleft(8)                # L vaut [8,5,7,3]
### Ajout à la fin de la FDE
L.append(0)                    # L vaut [8,5,7,3,0]
### Suppression au début de la FDE
a = L.popleft()                # a vaut 8 --- L vaut [5,7,3,0]
### Suppression à la fin de la FDE
b = L.pop()                    # b vaut 0 --- L vaut [5,7,3]
```

Le sujet est composé de deux exercices indépendants.

Exercice 1. Composantes fortement connexes

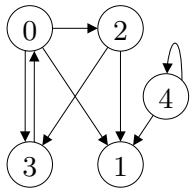

FIGURE 1

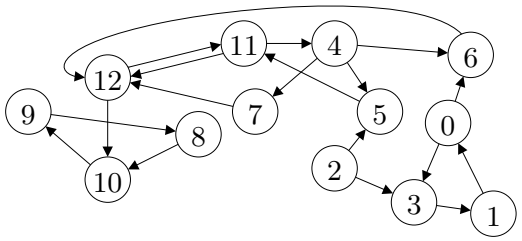

FIGURE 2

Dans cet exercice on s'intéresse à un graphe orienté $G = (S, A)$ où $S = [\![0, n-1]\!]$ avec $n \in \mathbb{N}$ le nombre de sommets. En Python, G est représenté par sa matrice d'adjacence « `M: list[list[int]]` ».

1. Donner la matrice d'adjacence « `MFig1: list[list[int]]` » pour le graphe de la figure 1.

Une ***composante fortement connexe*** de G est un ensemble non vide de sommets $S' \subset S$ tel que :

- ⋆ Pour tout $u \in S'$ et tout $v \in S'$, il existe un chemin de u à v et il existe un chemin de v à u.
- ⋆ Pour tout $u \in S'$ et tout $v \in S \setminus S'$, il n'existe pas de chemin de u à v, ou il n'existe pas de chemin de v à u.

On admet que l'ensemble des sommets peut toujours être partitionné comme une union disjointe de composantes fortement connexes :

$$S = \bigcup_{i=1}^{k} S_i \qquad \text{où} \qquad \begin{cases} \text{Pour tout } i,\ S_i \text{ est une composante fortement connexe.} \\ \text{Pour tout } i \neq j : S_i \cap S_j = \varnothing. \end{cases}$$

On admet également que pour tout sommet $u \in S$, il existe une unique composante fortement connexe $S' \subset S$ telle que $u \in S'$. Par exemple dans le graphe de la figure 2, la composante fortement connexe du sommet 0 est $S_1 = \{0, 1, 3\}$. En particulier, $2 \notin S_1$ puisqu'il n'existe pas de chemin entre 0 et 2 et $6 \notin S_1$ puisqu'il n'existe pas de chemin entre 6 et 0.

2. Donner toutes les composantes fortement connexes du graphe de la figure 2.

Afin de déterminer les composantes fortement connexes de G, on propose de construire pour chaque sommet $u \in S$ la liste de tous les sommets $v \in S$ tels qu'il existe un chemin de u à v dans G.

3. À l'aide d'un parcours en largeur, écrire une fonction de signature

```
accessibles(M: list[list[int]], u: int) -> dict[int: NoneType]
```

qui renvoie un dictionnaire dont les clés sont tous les sommets `v` tels qu'il existe un chemin entre `u` et `v` dans G.

4. (a) Supposons avoir défini « `Acc: list[dict[int: NoneType]]` » la liste suivante :

```
Acc = [accessibles(M, u) for u in range(len(M))]
```

Écrire une fonction de signature

```
CFC(Acc: list[dict[int: NoneType]], u: int) -> list[int]
```

qui prend en entrée la liste `Acc` ainsi qu'un sommet `u` $\in [\![$`0, n - 1`$]\!]$ et renvoie la liste de tous les sommets dans la composante fortement connexe de `u`.

(b) En déduire une fonction de signature

```
ListeCFC(M: list[list[int]]) -> list[list[int]]
```

qui renvoie la liste de toutes les composantes fortement connexes du graphe. Chaque composante fortement connexe devra apparaître une et une seule fois dans la liste renvoyée.

Exercice 2. Tri topologique

Monsieur Cosinus aide régulièrement monsieur Sinus à apprendre les gestes simples de la vie quotidienne. Aujourd'hui, la leçon porte sur l'ordre dans lequel il faut mettre ses vêtements afin de s'habiller correctement pour le bal. Voici deux exemples de règles :

- Il n'est pas possible de mettre sa cravate avant sa chemise.
- Les chaussures doivent être mises après le caleçon, le pantalon et les chaussettes.

Pour aider son ami à comprendre les différentes règles, M. Cosinus décide d'utiliser une représentation sous forme de graphe orienté (voir la figure 3). Dans ce graphe, chaque arc représente une contrainte : s'il existe un arc entre le sommet A et le sommet B, cela signifie que le vêtement A doit être mis avant le vêtement B. En particulier, les deux règles énoncées ci-dessus se déduisent bien du graphe de la figure 3.

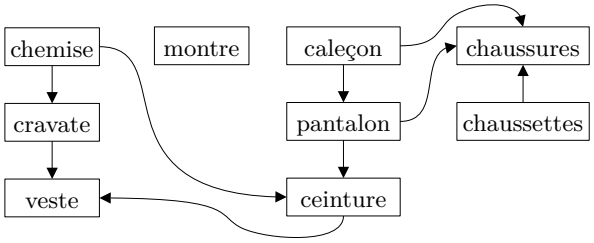

FIGURE 3

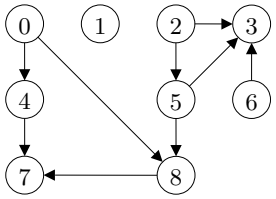

FIGURE 4

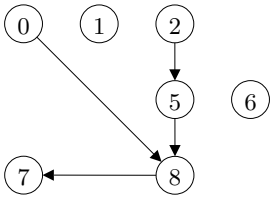

FIGURE 5

L'objectif de M. Sinus est, à partir de la figure 3, de déterminer l'ordre dans lequel mettre ses vêtements. Voici une possibilité :

(T_1) : chaussettes, caleçon, pantalon, chaussures, montre, chemise, ceinture, cravate, veste.

Afin de formaliser le problème, on considère un graphe orienté $G = (S, A)$ dont les sommets sont numérotés de 0 à $|S| - 1$ (la figure 4 montre une numérotation possible des sommets de la figure 3). Dans la suite, on ne fera pas de distinction entre un "sommet u" et le "numéro du sommet u". Notre but est de calculer un ***tri topologique*** du graphe, c'est à dire de déterminer une liste « `T: list[int]` » de taille $|S|$ contenant une et une seule fois chaque entier de $[\![0; |S| - 1]\!]$, et tel que si G contient l'arc (u, v) alors u apparaît strictement avant v dans `tab`. Par exemple, le tri topologique (T_1) correspond à la liste :

```
T1 = [6, 2, 5, 3, 1, 0, 8, 4, 7]
```

1. Représentation des graphes en Python

Listes de successeurs. En Python, un graphe orienté $G = (S, A)$ sera représenté par une liste contenant les listes de successeurs des sommets. Par exemple, le graphe de la figure 4 correspond à `GFig4`.

```
GFig4 = [[4,8], [], [3,5], [], [7], [3,8], [3], [], [7]]
```

Dans la suite, le type `list[list[int]]` sera noté `graphe`. Ainsi, une fonction de signature « `f(G: graphe) -> int` » est une fonction qui prend en entrée une liste `G` de type `list[list[int]]` représentant un graphe, et qui renvoie un entier.

Soit « `G: graphe` » un graphe avec `n` $\in \mathbb{N}$ sommets et « `T1: list[int]` » une liste. Pour que `T1` soit un tri topologique de `G`, trois conditions doivent être respectées :

(C1) `T1` est de taille `n`.

(C2) Tous les éléments de $[\![$`0, n - 1`$]\!]$ appartiennent à `T1`.

(C3) Pour tout sommet `u` $\in [\![$`0; n - 1`$]\!]$ et tout sommet `v` $\in [\![$`0; n - 1`$]\!]$, si `(u, v)` est un arc de `G`, alors l'indice de `u` dans `T1` est strictement inférieur à l'indice de `v` dans `T1`.

1. (a) Écrire une fonction de signature `test1(n: int, T1: list[int]) -> bool` qui renvoie un booléen indiquant si la condition (C1) est vérifiée.

(b) Quelle est la complexité de votre fonction ?

2. Écrire une fonction de signature `test2(n: int, T1: list[int]) -> bool` qui renvoie un booléen indiquant si la condition (C2) est vérifiée. La complexité de votre fonction devra être en $\mathcal{O}$(`max(n,len(T1))`).

3. Dans cette question, on suppose que `test1(n, T1)` et `test2(n, T1)` s'évaluent en `True`.

   (a) Écrire une fonction de signature `inverse(T1: list[int]) -> list[int]` qui renvoie une liste « `T2: list[int]` » de taille `n` telle que pour tout sommet `u`, `T2[u]` est l'indice de `u` dans `T1`. Par exemple :

   Si `T1 = [6, 2, 5, 3, 1, 0, 8, 4, 7]` alors `T2 = [5, 4, 1, 3, 7, 2, 0, 8, 6]`.

   On fera en sorte de ne parcourir qu'une seule fois les éléments de `T1`.

   (b) En déduire une fonction de signature `test3(G: graphe, T1: list[int]) -> bool` qui renvoie un booléen indiquant si la condition (C3) est vérifiée.

   (c) Donner en la justifiant la complexité de la fonction `test3`. On attend une justification détaillée.

4. Écrire une fonction de signature `estTriTopo(G: graphe, T1: list[int]) -> bool` qui renvoie un booléen indiquant si `T1` est un tri topologique de `G`.

Sous-graphes induits. On dit que G est un ***sous-graphe induit*** de G_0 si G peut être obtenu en supprimant certains sommets dans G_0. Formellement, $G = (S, A)$ est un sous-graphe induit de $G_0 = (S_0, A_0)$ si et seulement si :

$$S \subset S_0 \qquad \text{et} \qquad A = \Big\{(u, v) \in A_0 : u \in S, v \in S\Big\}.$$

Par exemple, le graphe de la figure 5 est un sous-graphe induit de la figure 4, il est obtenu en supprimant les sommets 3 et 4.

En Python, si le graphe initial G_0 contient n_0 sommets et est représenté par une variable « `G0: graphe` », alors un sous-graphe induit G est représenté par le couple `(G0, S)` où « `S: list[bool]` » est une liste de taille n_0 telle que `S[u]` vaut `True` si et seulement si `u` est un sommet de G. Par exemple, pour le graphe de la figure 5, on obtient `GFig5` :

```
GFig5 = (GFig4, [True, True, True, False, False, True, True, True, True])
```

Dans la suite, on dira qu'une variable `G` est de type `grapheInduit` si c'est un couple `(G0, S)` où « `G0: graphe` », « `S: list[bool]` » et `len(G0) = len(S)`.

5. Écrire une fonction de signature

   `supprSommets(G: grapheInduit, L: list[int]) -> NoneType`

   qui prend en entrée un graphe induit `G = (G0,S)` ainsi qu'une liste `L`, et modifie `S` pour supprimer de G tous les sommets appartenant à `L`. On pourra supposer sans le vérifier que `L` ne contient que des sommets de `G`. Votre fonction doit modifier `S` et ne rien renvoyer.

6. Écrire une fonction de signature `degEntrant(G: grapheInduit) -> list[int]` qui prend en entrée un graphe induit `G = (G0,S)` avec `n` sommets et renvoie une liste `D` de taille `n` tel que `D[u]` est le degré entrant du sommet `u` dans `G`. Si `u` n'est pas un sommet de `G`, `D[u]` sera fixé à `-1`. Par exemple, pour `GFig5`, on obtient :

   `[0, 0, 0, -1, -1, 1, 0, 1, 2]`

2. Calcul d'un tri topologique

Pour rappel, un ***circuit*** est un chemin de longueur strictement positive, dont les arcs sont distincts deux à deux et dont les sommets de départ et d'arrivée sont identiques.

7. Montrer que s'il existe un circuit dans un graphe G alors il n'existe pas de tri topologique pour G.

Dans la suite de cette partie, tous les graphes considérés ne contiennent pas de circuit.

Première méthode. Monsieur Sinus propose une méthode pour obtenir un tri topologique à partir du graphe de la figure 3. Il remarque tout d'abord que les sommets étiquetés par "chemise", "montre" , "caleçon" et "chaussettes" ont tous les quatre un degré entrant égal à 0. On peut donc les placer au début du tri topologique (dans un ordre arbitraire). En supprimant ces quatre sommets du graphe, les sommets "pantalon" et "cravate" ont pour degré entrant 0 et peuvent donc être placés à la suite dans le tri topologique. On continue ainsi jusqu'à ce que tous les sommets aient été supprimés du graphe. Avec cette procédure on obtient :

(T_2) : chemise, montre, caleçon, chaussettes, cravate, pantalon, chaussures, ceinture, veste.

8. Soit G_0 un graphe sans circuit et G un sous-graphe induit de G_0 avec au moins un sommet. Montrer qu'il existe au moins un sommet de G dont le degré entrant est 0.
9. En généralisant l'idée ayant permis de construire le tri topologique (T_2), écrire une fonction de signature `triTopo1(G: graphe) -> list[int]` qui revoie un tri topologique de `G`. On attend ici que vous exploitiez le type `grapheInduit` de la partie 1..

Première méthode (bis). Monsieur Cosinus pense que la solution de M. Sinus n'est pas optimale en terme de complexité. Il propose donc d'accélérer la recherche des sommets ayant un degré entrant égal à 0. Son idée est d'utiliser trois variables :

- Une liste « `T: list[int]` » initialement vide et qui contiendra le tri topologique à la fin de la procédure.
- Une liste « `D: list[int]` » contenant les degrés entrants des sommets dans le sous-graphe induit.
- Une file `F` contenant les sommets ayant un degré entrant égal à 0, mais n'ayant pas encore été ajoutés à `T`.

À chaque étape, on récupère le sommet u le plus ancien ayant été ajouté dans `F`. On supprime u de `F`, on l'ajoute à la fin de `T` et met à jour `D` en diminuant les degrés de tous les successeurs de u dans G. Par exemple, voici l'évolution des trois variables lors des quatre premières étapes de la procédure pour le graphe de la figure 4 :

	Étape 1	Étape 2	Étape 3	Étape 4
`T`	`[]`	`[0]`	`[0,1]`	`[0,1,2]`
`D`	`[0,0,0,3,1,1,0,2,2]`	`[0,0,0,3,0,1,0,2,1]`	`[0,0,0,3,0,1,0,2,1]`	`[0,0,0,2,0,0,0,2,1]`
`F`	`(0,1,2,6)`	`(1,2,6,4)`	`(2,6,4)`	`(6,4,5)`

10. Dans cette question, on s'intéresse à la procédure appliquée au graphe de la figure 4.
    (a) Donner le tableau pour les étapes 5, 6 et 7.
    (b) Quel tri topologique obtient-on à la fin de la procédure ?
11. Écrire une fonction de signature `triTopo2(G: graphe) -> list[int]` qui renvoie un tri topologique du graphe en entrée en suivant cette méthode. On garantira une complexité en $\mathcal{O}(n+m)$ où n est le nombre de sommets dans G et m le nombre d'arcs. Pour `F`, on utilisera une file à double extrémité du module `collections` (voir le début de l'énoncé).

Deuxième méthode. Madame Tangente, l'amie en commun de M. Cosinus et de M. Sinus, pense pouvoir utiliser un parcours en profondeur pour calculer un tri topologique. Voici pour commencer un pseudo-code décrivant le parcours en profondeur à partir d'un sommet u :

Algorithme 1 – $PP(u)$ – Parcours en profondeur à partir du sommet u

```
# Début de traitement du sommet u
pour chaque sommet v voisin de u faire
    si v a déjà été rencontré lors d'un parcours précédent alors
        ne rien faire
    sinon
        appeler récursivement PP(v)
    fin si
fin pour
# Fin de traitement du sommet u
```

Étant donné qu'un appel à la procédure PP n'atteint pas forcément tous les sommets du graphe, il est parfois nécessaire d'effectuer plusieurs parcours en profondeur. La procédure principale que nous utiliserons est la suivante :

Algorithme 2 – Procédure principale

pour chaque sommet u du graphe **faire**
  si u a déjà été rencontré lors d'un parcours précédent **alors**
    ne rien faire
  sinon
    appeler $PP(u)$
  fin si
fin pour

Appliquons l'algorithme 2 avec le graphe $G = (S, A)$ de la figure 4. Voici les différents évènements qui se produisent : début de traitement du sommet 0, début de traitement du sommet 4, début de traitement du sommet 7, fin de traitement du sommet 7, fin de traitement du sommet 4, début de traitement du sommet 8, fin de traitement du sommet 8, fin de traitement du sommet 0. La procédure principale lance ensuite d'autres appels à la fonction PP : à partir du sommet 1, puis à partir du sommet 2 et enfin à partir du sommet 6. Ces évènements peuvent maintenant être numérotés de 1 à $2|S|$ en suivant l'ordre dans lequel ils se produisent. On obtient :

Sommet u	0	1	2	3	4	5	6	7	8
$deb(u)$	1	9	11	12	2	14	17	3	6
$fin(u)$	8	10	16	13	5	15	18	4	7

où $deb(u)$ et $fin(u)$ sont les instants de début et de fin de traitement du sommet u. On appelle ***ordre préfixe*** (resp. ***ordre postfixe***) la liste des sommets triés par ordre croissant de début (resp. fin) de traitement :

Ordre préfixe : (0,4,7,8,1,2,3,5,6), Ordre postfixe : (7,4,8,0,1,3,5,2,6).

12. Supposons avoir exécuté l'algorithme 2 sur un graphe $G = (S, A)$ ne contenant pas de circuit.
    (a) Montrer qu'il n'existe pas d'arc $(u, v) \in A$ tel que $deb(v) < deb(u)$ et $fin(v) > fin(u)$.
    (b) En déduire comment obtenir un tri topologique en utilisant l'ordre postfixe. Le montrer.
13. Écrire une fonction de signature `triTopo3(G: graphe) -> list[int]` qui renvoie le tri topologique obtenu avec la méthode trouvée à la question précédente.

3. Génération de tous les tris topologiques

Mademoiselle Factorielle, la belle-fille de M. Cosinus, souhaite générer exhaustivement tous les tris topologiques d'un graphe pour pouvoir les compter.

14. (a) Écrire une fonction récursive de signature

```
perm(L: list[int]) -> list[list[int]]
```

qui renvoie une liste contenant toutes les permutations de `L`. Par exemple pour `L = [0,1,2]` on obtient (l'ordre des sous-listes n'a pas d'importance) :

```
[[0,1,2], [1,0,2], [0,2,1], [2,0,1], [1,2,0], [2,1,0]]
```

(b) En déduire une fonction de signature

```
compterTT(G: graphe) -> int
```

qui renvoie le nombre de tris topologiques pour `G`.