Problème 1. Ordres topologiques

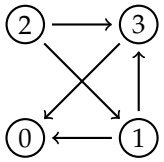
Dans ce problème, on considère des graphes *orientés* $G = (S, A)$, où S est l'ensemble des *sommets* et $A \subset S \times S$ est l'ensemble des *arcs*. Un *voisin*, ou *successeur* d'un sommet s est un sommet $t \in S$ tel que $(s, t) \in A$. Similairement, un *prédécesseur* d'un sommet s est un sommet t tel que $(t, s) \in A$. On note $|S|$ et $|A|$ le nombre de sommets et d'arcs du graphe, respectivement. Dans tout le problème, on identifiera l'ensemble S à l'ensemble $\{0, \dots, n-1\}$, où $n = |S|$. Le *degré entrant* d'un sommet est son nombre de prédecesseurs.
Soit $G = (S, A)$ un graphe orienté à n sommets. Un *ordre topologique* sur G est une fonction $o : S \to \{0, \dots, n-1\}$ qui vérifie les propriétés suivantes :

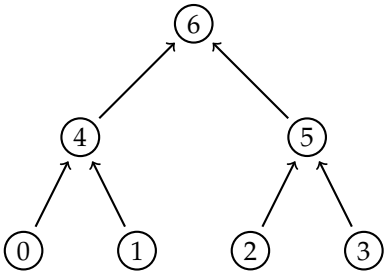
— o est bijective : pour $0 \leq i < n$, il existe un unique sommet $v \in S$ tel que $o(v) = i$;
— pour toute arête $(u, v) \in A$, $o(u) < o(v)$.

Représentations Dans les algorithmes, les graphes seront représentés par *listes d'adjacence* : la représentation d'un graphe G est un liste `G` de listes, telle que `G[i]` soit la liste des voisins du sommet i (sans ordre particulier). Par exemple, les graphes de la question 1.1 ci-dessous peuvent être représentés par les listes `[[], [0, 3], [3, 1], [0]]` et `[[4], [4], [5], [5], [6], [6], []]`.
Un ordre topologique sera représenté par une liste `o` telle que pour tout i, `o[i]` contienne la valeur $o(i)$. Par exemple, si $S = \{0, 1, 2\}$ et que o est défini par $o(0) = 2$, $o(1) = 1$ et $o(2) = 0$, on aura `o = [2, 1, 0]`.

Question 1.1.

(a) Donner pour chacun des deux graphes suivants un ordre topologique.





(b) Montrer qu'un graphe qui possède un cycle n'admet pas d'ordre topologique.
(c) Montrer qu'un graphe admet un ordre topologique si et seulement s'il est sans cycle.
(d) Écrire un algorithme `Inverse` qui prend en entrée une liste o représentant un ordre topologique, et renvoie la liste $[o^{-1}(0), o^{-1}(1), \dots, o^{-1}(n-1)]$, où o^{-1} représente la réciproque de la fonction o.

Question 1.2. Utilitaires

(a) Écrire un algorithme `nb_sommets` et un algorithme `nb_arcs` qui calculent respectivement le nombre de sommets et le nombre d'arcs d'un graphe dans la représentation décrite ci-dessus.
(b) Écrire un algorithme `retourne` qui prend en entrée un graphe G et renvoie le graphe G' obtenu en retournant chaque arc dans G : (j, i) est un arc de G' si et seulement si (i, j) est un arc de G. Exprimer la complexité de l'algorithme en fonction du nombre de sommets et/ou d'arcs de G.
(c) Écrire un algorithme `degres_entrants` qui prend en entrée un graphe G et renvoie la liste des degrés entrants de chaque sommet. Exprimer la complexité de l'algorithme en fonction du nombre de sommets et/ou d'arcs de G.
(d) Écrire un algorithme `entrees` qui prend en entrée un graphe G et renvoie la liste des sommets de degré entrant nul.

Question 1.3. Calcul d'ordre topologique L'objectif de cette question est de calculer, étant donné un graphe G, un ordre topologique pour G s'il en existe un. On utilise l'algorithme suivant. On maintient la liste des *entrées* du graphe, c'est-à-dire les sommets de degré entrant nul. Une étape de l'algorithme consiste à traiter un des sommets de la liste des entrées, de la manière suivante :

1. on le supprime de la liste des entrées ;
2. on lui attribue la plus petite valeur non encore attribuée dans l'ordre topologique ;
3. on supprime tous les arcs qui partent de ce sommet ;
4. on recalcule la liste des entrées du nouveau graphe.

L'algorithme s'arrête lorsque la liste des entrées restantes est vide.

(a) Quels sont les différents cas possibles lorsque l'algorithme s'arrête, et que doit-on retourner en fonction de ces cas ?

(b) Montrer par récurrence sur le nombre d'arcs du graphe que l'algorithme est correct.

(c) Écrire un algorithme `supprime_arc` qui étant donné un graphe G et un arc (i, j) renvoie le graphe G' obtenu en supprimant (i, j) de G. *Attention, l'algorithme ne doit pas modifier G lui-même.*

(d) Compléter l'algorithme `ordre_topologique` qui prend en entrée un graphe G et renvoie un ordre topologique pour G s'il en existe un, et une liste vide sinon.

```
def ordre_topologique(G):
    n = nb_sommets(G)
    degres = degres_entrants(G)
    entrees = [i for i in range(n) if ... ]
    H = G.copy()                              # Copie de G
    v = 0
    o = [-1] * len(G)                         # [-1, -1, ..., -1]

    while len(entrees) > 0:
        s = entrees.pop() # supprime le premier élément et le renvoie
        o[s] = ...
        v = ...

        for j in H[s]:
            degres[j] = ...
            if degres[j] == 0:
                ...
        H[s] = ...

    if min(o) == -1:
        return ...

    return o
```

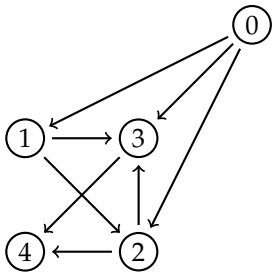
(e) Démontrer que l'algorithme a pour complexité $O(|S| + |A|)$.

Question 1.4. Tri linéaire Écrire un algorithme `tri_successeurs` qui prend en entrée un graphe G et un ordre topologique o pour G, et renvoie une nouvelle description de G dans laquelle les listes d'adjacence sont triées selon l'ordre o. La complexité de l'algorithme doit être $O(|S| + |A|)$. *Il n'est donc pas possible d'utiliser un tri par comparaisons.*

Question 1.5. Fermeture réflexive transitive La *fermeture réflexive transitive* d'un graphe sans cycle $G = (S, A)$ est le graphe $G^\star = (S, A^\star)$ où $(i, j) \in A^\star$ si et seulement s'il existe un chemin de i vers j dans A : $(i, j) \in A^\star$ si et seulement s'il existe $s_0, \ldots, s_k$ dans V avec $s_0 = i$, $s_k = j$ et tels que pour tout $\ell \in \{0, \ldots, k-1\}$, $(s_\ell, s_{\ell+1}) \in A$. Le cas $k = 0$ implique que pour tout $i \in V$, $(i, i) \in A^\star$.
Dans la suite, les graphes considérés sont sans cycle. De plus, on suppose que les sommets sont triés de telle sorte que la fonction $o : i \mapsto i$ est un ordre topologique. Enfin, on suppose également que les listes d'adjacence sont triées selon l'ordre topologique (qui est donc l'ordre naturel sur les entiers).
On considère l'algorithme `fermeture` décrit ci-après, qui prend en entrée la description d'un graphe G par listes d'adjacence.

```
1  def fermeture(G):
2      n = len(G)
3      Q = [False] * n              # Liste de n booléens False
4      R = [[i] for i in range(n)]  # Liste [[0], ..., [n-1]]
5      for i in reversed(range(n)): # i va de n-1 à 0
6          Q[i] = True
7          for j in G[i]:
8              if not Q[j]:
9                  for k in R[j]:
10                     if not Q[k]:
11                         Q[k] = True
12                         R[i] = [k] + R[i]
13         for k in R[i]:
14             Q[k] = False
15     return R
```

(a) Pour le graphe suivant, donner les valeurs de R et Q à l'entrée de la boucle de la ligne 5.



(b) Montrer qu'à l'entrée de la boucle de la ligne 5, $R[j]$ contient la liste d'adjacence du sommet j dans le graphe $G^\star$ pour tout $j > i$.

(c) En déduire que l'algorithme renvoie une description du graphe $G^\star$.

On note A^- le sous-ensemble des arcs $(i, j) \in A$ tels qu'il n'existe pas de chemin de longueur supérieure ou égale à 2 entre i et j dans G. Par exemple, dans le graphe précédent, $A^- = \{(0,1), (1,2), (2,3), (3,4)\}$ mais $(1,3) \notin A^-$ bien que $(1,3) \in A$ car $1 \to 2 \to 3$ est un chemin de longueur 2 entre 1 et 3.

(d) Montrer que $|A^-|$ est borné par le nombre de sommets internes de G, c'est-à-dire par le nombre de sommets de degré entrant non nul.

(e) Montrer que $|A^\star| \leq |S||A^-| + |S|$.

(f) Montrer qu'à la ligne 8, la condition `not Q[j]` est satisfaite si et seulement si l'arc (i, j) appartient à A^-.

(g) En déduire que la complexité de l'algorithme est $O(|S|^3)$.