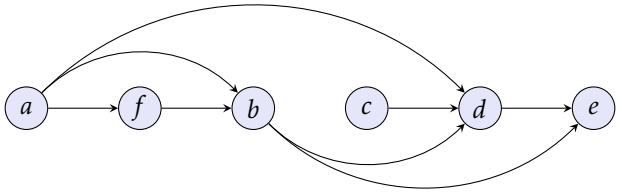
Corrigé : graphes ordonnés

Partie I. Tri topologique

Question 1. Considérons un graphe ordonné G. ainsi qu'un tri topologique $(s_0,\dots,s_{n-1})$ de ses sommets. Un éventuel cycle dans G prendrait la forme $(s_{i_1},\dots,s_{i_k})$ avec $i_1 < i_2 < \cdots < i_k$ avec $k \geqslant 3$ et $i_1 = i_k$, ce qui ne ce peut. G est donc acyclique.

Question 2. G_1 est ordonné, par exemple par l'intermédiaire du tri topologique (a,f,b,c,d,e) :



Remarque. il y a quatre tris topologiques possibles : (a,f,c,b,d,e), (c,a,f,b,d,e), (a,c,f,b,d,e), (a,f,b,c,d,e).

Question 3. Soit G un graphe de cardinal n dans lequel tout sommet possède au moins un successeur. Il est alors possible de construire une suite $(s_0,\dots,s_n)$ de $n+1$ sommets tels que pour tout $i \in [\![0,n-1]\!]$, $(s_i,s_{i+1}) \in A$.
Mais puisque $\operatorname{card} S = n$ il existe nécessairement $i < j$ tel que $s_i = s_j$ et alors $(s_i,\dots,s_j)$ est un cycle.
Par contraposée, dans un graphe acyclique il existe nécessairement au moins un sommet sans successeur.

Question 4. Montrons par récurrence sur n que tout graphe acyclique est ordonné :

- si $n = 1$ c'est évident ;
- si $n \geqslant 2$, considérons un sommet sans successeur et nommons-le s_{n-1}. Considérons maintenant le graphe G′ obtenu en supprimant de G le sommet s_{n-1} ainsi que toutes les arêtes menant à s_{n-1}. Le graphe G′ est toujours acyclique donc ordonné par hypothèse de récurrence. En notant $(s_0,\dots,s_{n-2})$ un tri topologique des sommets de G′ on construit un tri topologique $(s_0,\dots,s_{n-2},s_{n-1})$ de G, qui est donc ordonné. La récurrence se propage.

Question 5.

```
def supprime(G, s):
    Gprime = {}
    for x in G:
        if x != s:
            voisins = []
            for y in G[x]:
                if y != s:
                    voisins.append(y)
            Gprime[x] = voisins
    return Gprime
```

Question 6. Compte tenu de la question 4, une programmation récursive est adaptée.

```
def ordonne(G):
    n = len(G)
    if n == 0:
        return []
    for s in G:
        if G[s] == []:
            Gprime = supprime(G, s)
            ordre = ordonne(Gprime)
            ordre.append(s)
            return ordre
```

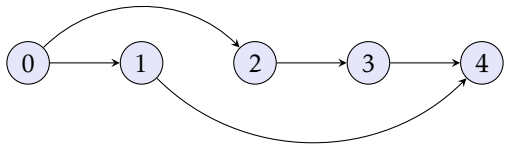
Partie II. Plus long chemin dans un graphe acyclique

Question 7. Raisonnons par récurrence sur n :

- si $n = 1$ c'est évident ;
- si $n > 1$, supposons le résultat acquis jusqu'au rang $n-1$.
  Par hypothèse le sommet 0 possède au moins un successeur $s \in [\![1, n-1]\!]$. Considérons le graphe G′ obtenu en supprimant les sommets $0, \ldots, s-1$ ainsi que les arêtes qui partent de ces sommets. G′ est toujours ordonné et tous les sommets restants hormis le dernier ont encore au moins un successeur, donc par hypothèse de récurrence il existe un chemin menant de s à $n-1$ dans G′, et donc un chemin menant de 0 à $n-1$ en passant par s dans G. La récurrence se propage.

Un algorithme glouton

Question 8. Notons (s_k) la suite de sommets construite à partir de cet algorithme. On a $s_0 = 0$ et puisque G est ordonné, pour tout entier k, $s_k < s_{k+1}$. Le nombre de sommets étant fini, il existe $k \in [\![2, n-1]\!]$ tel que $s_k = n-1$: l'algorithme se termine.
Cet algorithme ne donne cependant pas une solution optimale, puisque pour le graphe suivant il construit un chemin de longueur deux alors qu'un chemin de longueur trois existe :



Question 9.

```
def glouton(G):
    n = len(G)
    s = 0
    longueur = 0
    while s < n - 1:
        s = min(G[s])
        longueur += 1
    return longueur
```

Programmation dynamique et mémoïsation

Question 10. En notant G[k] l'ensemble des successeurs de k, on a $\ell(k) = 1 + \max_{i \in G[k]} \ell(i)$.

Question 11.

```
dico = {}
def longueurMax(G, k):
    n = len(G)
    if k not in dico:
        if k == n - 1:
            dico[n - 1] = 0
        else:
            dico[k] = 1 + max([longueurMax(G, i) for i in G[k]])
    return dico[k]
```

Plus long chemin

Question 12.

```
def longueursMax(G):
    n = len(G)
    longMax = [0 for _ in range(n)]
    for k in range(n - 2, -1, -1):
        longMax[k] = 1 + max([longMax[i] for i in G[k]])
    return longMax
```

Question 13.

```
1  def cheminMax(G):
2      n = len(G)
3      longMax = longueursMax(G)
4      s = n - 1
5      chemin = [s]
6      while s != 0:
7          k = s - 1
8          while not (s in G[k] and longMax[k] == longMax[s] + 1):
9              k -= 1
10         s = k
11         chemin = [s] + chemin
12     return chemin
```

Tant que le sommet s n'est pas égal à 0, les lignes 7-8-9 recherchent un sommet k de G vérifiant $s \in G[k]$ et $\ell(k) = \ell(s) + 1$ (par définition de ℓ un tel sommet existe forcément).

Partie III. Chemin de poids maximal dans un graphe pondéré

Question 14. On a ici $p_m(k) = \max\limits_{i \geqslant k+1}\Big(g(k,i) + p_m(i)\Big)$. D'où la fonction :

```
def poidsMax(G):
    n = len(G)
    pm = [0 for _ in range(n)]
    for k in range(n - 2, -1, -1):
        pm[k] = max([G[k][i] + pm[i] for i in range(k + 1, n)])
    return pm[0]
```

Il s'agit d'un algorithme de complexité temporelle en $O(n^2)$.

Question 15.

a) On a $M(n-1,r) = \begin{cases} 0 & \text{si } r = 0 \\ -\infty & \text{si } r \geqslant 1 \end{cases}$ et $M(k,0) = \begin{cases} 0 & \text{si } k = n-1 \\ -\infty & \text{si } k < n-1 \end{cases}$.

b) Pour tout $k \in [\![0, n-2]\!]$, pour tout $r \in [\![1, n-1]\!]$, $M(k,r) = \max\limits_{i \geqslant k+1}\Big(g(k,i) + M(i,r-1)\Big)$.

c) Compte tenu de la relation de dépendance on obtient :

```
def poidsMaximum(G):
    n = len(G)
    M = [[-inf for r in range(n)] for k in range(n)]
    M[n - 1][0] = 0
    for r in range(1, n):
        for k in range(n - 2, -1, -1):
            M[k][r] = max([G[k][i] + M[i][r - 1] for i in range(k + 1, n)])
    return M
```