Graphes (Graphs)
Implémentations et parcours

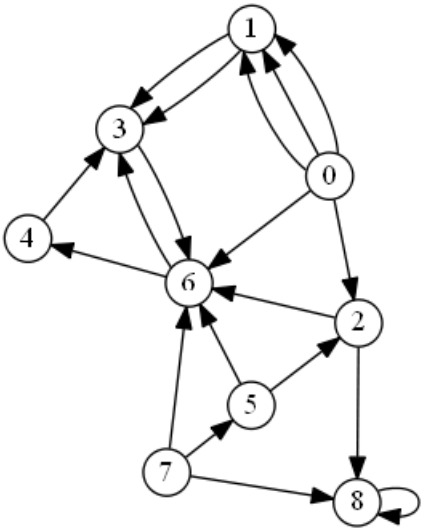


FIGURE 1 – Digraph G'_1

`files/tuto_digraph1multi.gra`

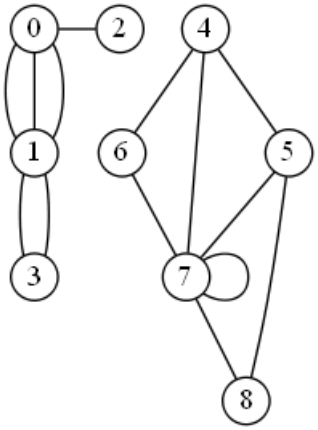


FIGURE 2 – Graph G'_2

`files/tuto_graph2multi.gra`

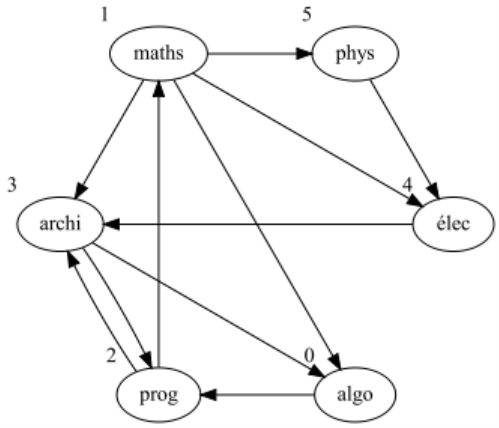


FIGURE 3 – Graph G_3

`files/digraph_subjects.gra`

1 Représentations / Implémentations

Exercice 1.1 (`GraphMat` : Matrice d'adjacence)

On utilise pour cette première implémentation, la représentation par matrice d'adjacence.

1. Quelles sont les différences, pour l'implémentation, lorsque le graphe est orienté ou non orienté, valué ou non, possède des liaisons simples ou multiples ?
2. Donner les matrices d'adjacence représentant les graphes des figures 1 et 2.
3. Sachant que l'on veut pouvoir utiliser le même type pour représenter un graphe qu'il soit orienté ou non, simple ou 1-graphe, ou encore un multigraphe ou p-graphe, que doit contenir l'implémentation ?

Exercice 1.2 (`Graph` : Listes d'adjacence)

1. Quelle est l'autre manière de représenter / implémenter un graphe ?
2. Quelles sont les différences pour l'implémentation lorsque le graphe est orienté ou non orienté, possède des liaisons multiples ?
3. Donner les représentations des graphes des figures 1 et 2.
4. Sachant que l'on veut pouvoir utiliser le même type pour représenter n'importe quel type de graphe : orienté ou non, simple (ou 1-graphe) ou multigraphe (ou p-graphe) ; que doit contenir l'implémentation ?

Exercice 1.3 (Degrés)

1. Écrire une fonction qui construit le vecteur (une liste en Python) des degrés de tous les sommets d'un graphe non orienté représenté par matrice d'adjacence.

   Exemple avec `G2mat` le graphe de la figure 2 (G'_2) :

```
>>> degrees(G2mat)
[4, 5, 1, 2, 3, 3, 2, 6, 2]
```

2. Écrire une fonction qui construit deux vecteurs (listes en Python) *in* et *out* qui contiendront respectivement les demi-degrés intérieurs et extérieurs de tous les sommets d'un graphe orienté représenté par listes d'adjacence.

   Exemple avec `G1` le graphe de la figure 1 (G'_1)

```
>>> in_out_degrees(G1)
([0, 3, 2, 4, 1, 1, 5, 0, 3], [5, 2, 2, 1, 1, 2, 2, 3, 1])
```

Exercice 1.4 (`dot`)

Écrire les fonctions qui construisent la représentation au format `dot` (une chaîne de caractères) d'un graphe, qu'il soit orienté ou non dans les deux implémentations.

Exemples :

- Graphe G'_1 (figure 1)

```
>>> print(dot(G1))
digraph {
  0 -> 1
  0 -> 1
  0 -> 1
  0 -> 6
  0 -> 2
  1 -> 3
  1 -> 3
  2 -> 6
  2 -> 8
  3 -> 6
  4 -> 3
  5 -> 2
  5 -> 6
  6 -> 3
  6 -> 4
  7 -> 6
  7 -> 5
  7 -> 8
  8 -> 8
}
```

- Graphe G'_2 (figure 2)

```
>>> print(dot(G2))
graph {
  1 -- 0
  1 -- 0
  1 -- 0
  2 -- 0
  3 -- 1
  3 -- 1
  5 -- 4
  6 -- 4
  7 -- 4
  7 -- 5
  7 -- 6
  7 -- 7
  8 -- 5
  8 -- 7
}
```

- **Bonus** Graphe G_3 (figure 3)

```
>>> print(dot(G3))
digraph  G {
 0[label = "algo"]
 0 -> 2
 1[label = "maths"]
 1 -> 0
 1 -> 3
 1 -> 4
 1 -> 5
 2[label = "prog"]
 2 -> 1
 2 -> 3
 3[label = "archi"]
 3 -> 2
 3 -> 0
 4[label = "elec"]
 4 -> 3
 5[label = "phys"]
 5 -> 4
}
```

Exercice 1.5 (Load - Save)

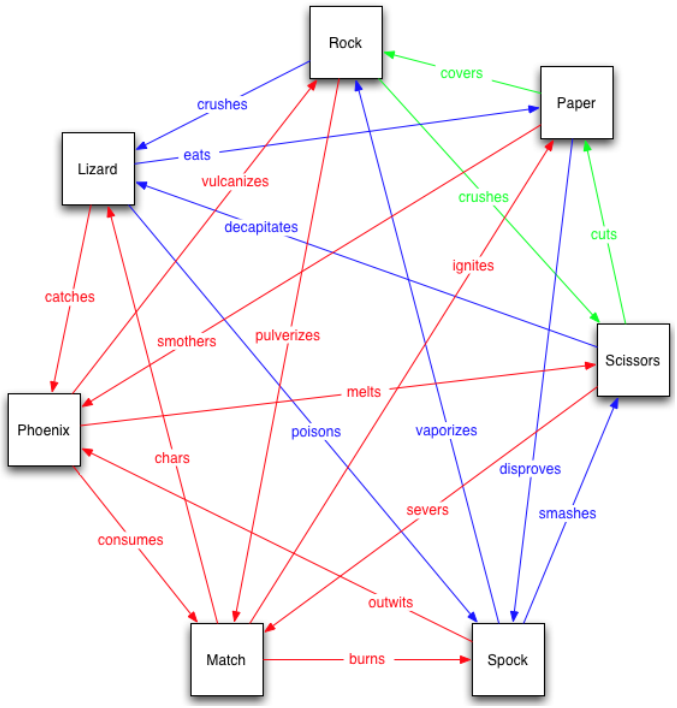
Notre format de fichier `GRA` est un fichier texte (voir `files/*.gra`) contenant :

- des lignes **optionnelles** commençant par `#` représentant des informations sur le graphes (bonus)
- une ligne contenant 0 ou 1 : 0 pour non orienté, 1 pour orienté
- une ligne contenant l'ordre du graphe
- une suite de lignes contenant les liaisons : deux numéros de sommets séparés par un espace

Étudier les fonctions données (dans `algopy/graph|graphmat.py`) qui chargent / enregistrent un graphe depuis / dans un fichier `".gra"` dans les deux implémentations.

`files/rpssmpl.gra` (extract) :

```
#name: RPSSMPL
#labels: rock,paper,scissors,Spock,match,phoenix,lizard
1
7
0 2
0 4
0 6
1 0
1 3
1 5
2 1
2 4
2 6
3 0
3 2
3 5
4 1
4 3
4 6
5 0
5 2
5 4
6 1
6 3
6 5
```



2 Parcours

Notes : Sauf indications particulières, dans la suite nous utiliserons des **graphes simples et des 1-graphes sans boucles**. Les exemples utilisés ici seront les graphes G_1 (issu de G'_1) et G_2 (issu de G'_2).

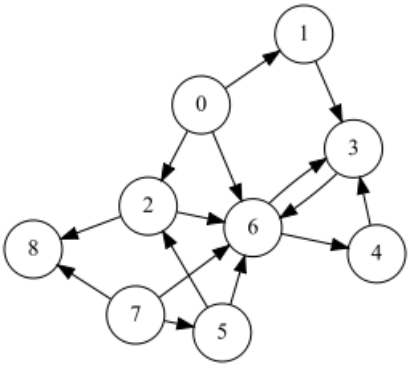


FIGURE 4 – Digraph G_1
`files/tuto_digraph1.gra`

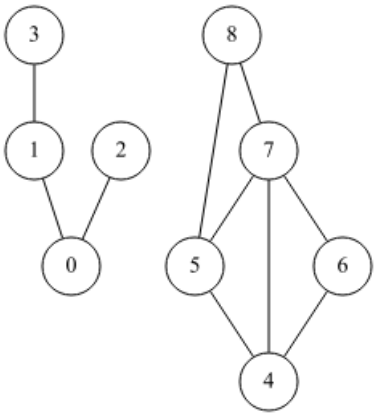


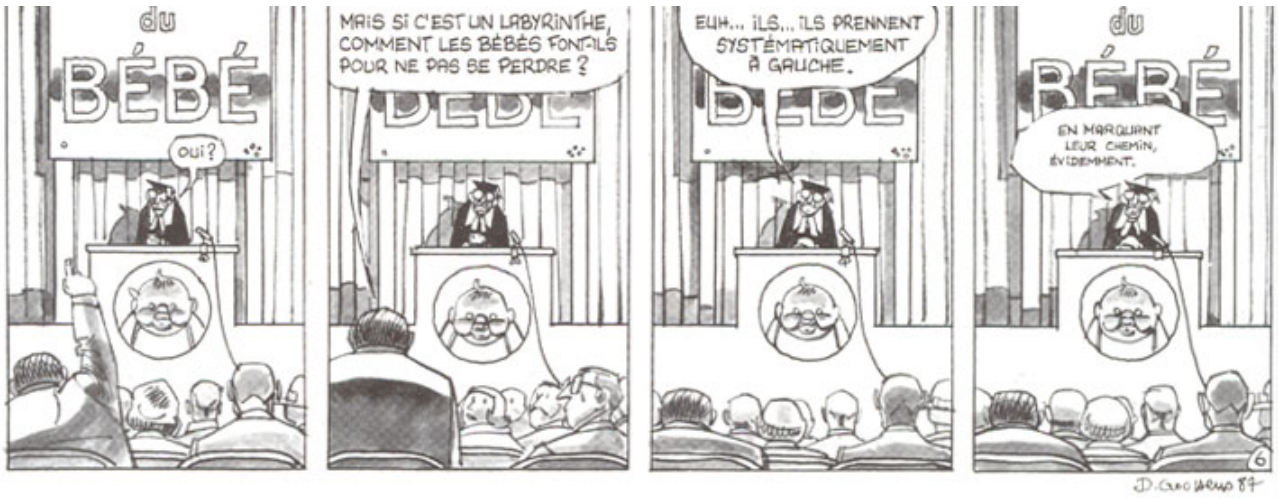
FIGURE 5 – Graph G_2
`files/tuto_graph2.gra`

Exercice 2.1 (BFS : Parcours largeur (Breadth-first search))

1. Donner le principe de l'algorithme de parcours largeur. Comparer avec le parcours d'un arbre général.
2. Donner les forêts couvrantes obtenues lors des parcours largeur complets des graphes G_1 et G_2 à partir du sommet 0 puis à partir du sommet 7 (les sommets sont choisis en ordre croissant).
3. Comment stocker la forêt couvrante de manière linéaire ?
4. Écrire les fonctions de parcours largeur :
   (a) qui affiche les sommets dans l'ordre de rencontre, un "arbre" par ligne, avec l'implémentation par matrice d'adjacence ;
   (b) qui construit le vecteur représentant la forêt couvrante, avec l'implémentation par listes d'adjacence.

Exercice 2.2 (DFS : Parcours profondeur (Depth-first search))

1. Donner le principe de l'algorithme récursif du parcours profondeur. Comparer avec le parcours d'un arbre général.
2. Donner les forêts couvrantes obtenues lors des parcours profondeur des graphes G_1 et G_2 à partir du sommet 0 puis du sommet 7 (les sommets sont choisis en ordre croissant).
3. Écrire les fonctions de parcours profondeur qui affichent les sommets dans l'ordre de première rencontre (préfixe), un "arbre" par ligne, pour les deux implémentations.
   Que faut-il changer pour construire la forêt couvrante ?
4. **Graphes non orientés (Undirected graphs)**
   (a) Quels sont les différents types d'arcs rencontrés lors du parcours profondeur d'un graphe non orienté ? Classer les arcs du parcours profondeur de G_2 effectué en question 2 et ajouter les arcs manquants à la forêt.
   (b) Que faut-il ajouter au parcours profondeur pour repérer les différents types d'arcs ?
   (c) Écrire la fonction du parcours profondeur d'un graphe non orienté. Indiquer au cours du parcours, le type des arcs rencontrés.
5. **Graphes orientés (Digraphs)**
   (a) Quels sont les différents types d'arcs rencontrés lors d'un parcours profondeur ? Comment les reconnaître ?
   Ajouter à la forêt couvrante du parcours profondeur de G_1 effectué en question 2 les arcs manquants, avec une légende explicite.
   (b) On utilise la notion d'ordre préfixe de visite (première rencontre) et ordre suffixe de visite (dernière rencontre). En numérotant les sommets suivant ces deux ordres avec un unique compteur, écrire les conditions de classification des arcs.
   (c) Écrire la fonction du parcours profondeur d'un graphe orienté.Indiquer au cours du parcours, le type des arcs rencontrés.



Graphes (Graphs)
Applications

Exercice 3.1 (Connect me)

Écrire la fonction `components` qui détermine les composantes connexes d'un graphe non orienté.

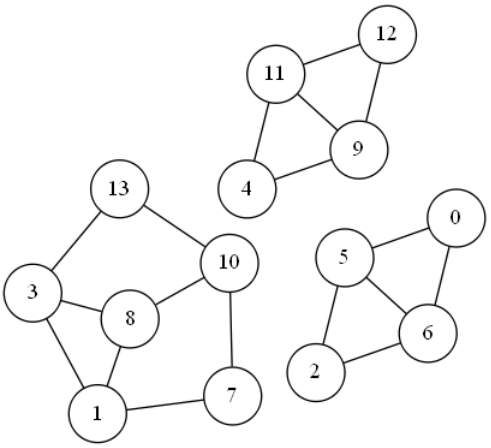


Figure 1 – Graphe `G_3cc`

La fonction retourne un couple :
- Le nombre de composantes connexes du graphe ;
- le vecteur des composantes : pour chaque sommet le numéro de la composante à laquelle il appartient.

Exemple d'application, avec `G_3cc` le graphe de la figure 1 :

```
>>> components(G_3cc)
(3, [1, 2, 1, 2, 3, 1, 1, 2, 2, 3, 2, 3, 3, 2])
```

`files/graph_3comp.gra`

Exercice 3.2 (That's the way)

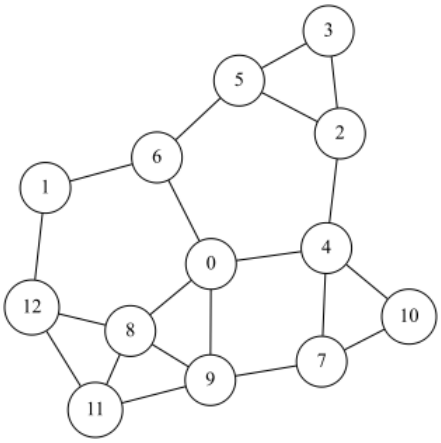


Figure 2 – `Gc: files/graph_center.gra`

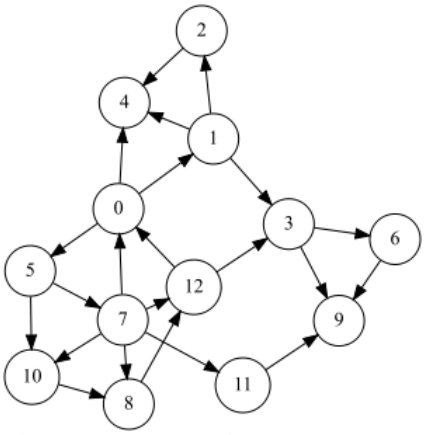


Figure 3 – `Gp: files/digraph_path.gra`

1. Comment trouver un chemin (ou une chaîne) entre deux sommets donnés dans un graphe ? Donner au moins deux méthodes différentes, les comparer. Prendre comme exemples les chaînes entre 0 et 1 dans `Gc` (figure 2) et les chemins entre 7 et 3 dans `Gp` (figure 3).
2. Écrire les deux fonctions qui cherchent un chemin entre deux sommets (différents). Si un chemin a été trouvé, il devra être retourné (une liste de sommets).

Exercice 3.3 (Influenceurs – *S3-P3 2022*)

Afin de maximiser l'impact d'une publication, une entreprise recherche dans un graphe social les *influenceurs* : l'ensemble de personnes qui permettra de diffuser une information le plus rapidement possible.

Définitions :

- La **distance** entre deux sommets d'un graphe est le nombre d'arêtes d'une **plus courte chaîne** entre ces deux sommets.
- On appelle **excentricité** d'un sommet x dans un graphe $G =< S, A >$ la quantité :

$$\text{exc}(x) = \max_{y \in S}\{distance(x, y)\}$$

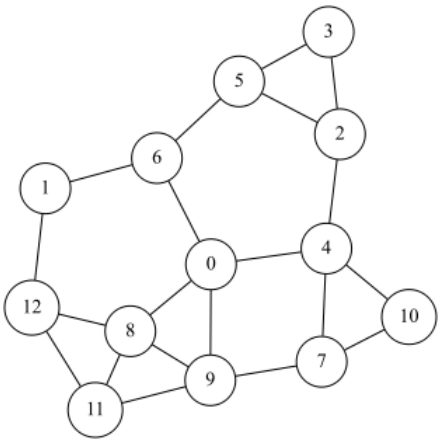
Une manière simple de déterminer les *influenceurs* est de chercher ceux qui sont à une distance minimale de n'importe quelle autre personne : dont l'excentricité dans le graphe est minimum.

Écrire la fonction `influencers(`G`)` qui retourne la liste des *influenceurs* du graphe connexe G (les sommets qui sont d'excentricité minimum).

Pour le graphe Gc (ci-contre) :
- Les sommets 0, 4 et 6 ont pour excentricité 3.
- Les sommets 3 et 11 ont pour excentricité 5.
- Les sommets restants ont pour excentricité 4.

L'excentricité minimum dans Gc est donc 3 et les *influenceurs* sont les sommets 0, 4 et 6.

```
>>> influencers(Gc)
[0, 4, 6]
```



Graphe `Gc`

Exercice 3.4 (Colors – *S3-P3 2021*)

En théorie des graphes, *colorer un graphe* signifie attribuer une couleur à chacun de ses sommets de manière à ce que deux sommets adjacents soient de couleurs différentes. Le nombre minimal de couleurs pour colorer un graphe est appelé *nombre chromatique* $\chi(G)$.

1. Donner une définition formelle du nombre chromatique $\chi(G)$ du graphe non orienté $G =< S, A >$.
2. Quel est le *nombre chromatique* (le nombre minimum de couleurs, $\chi(G_i)$) des graphes suivants ?
   Les couleurs seront représentées par des entiers. Pour chaque graphe, remplir le tableau des couleurs : des entiers dans $[1, \chi(G_i)]$. `files/graph_colors_i.gra`

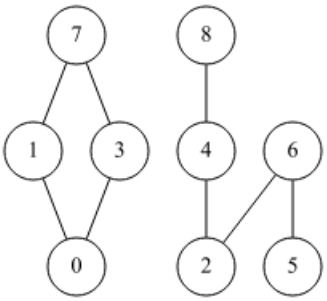


FIGURE 4 – G_1

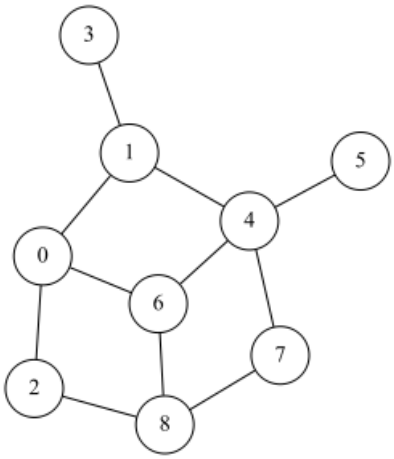


FIGURE 5 – G_2

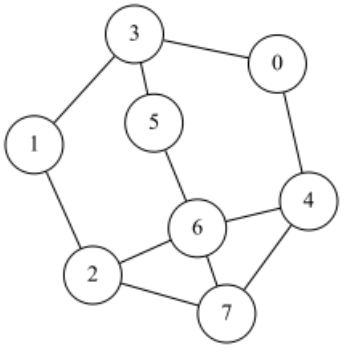


FIGURE 6 – G_3

3. Écrire la fonction `two_coloring(`G`)` qui vérifie si $\chi(G) = 2$ (le graphe peut être coloré avec 2 couleurs).

Exercice 3.5 (I want to be tree)

Définition :
Un **arbre** est un graphe **connexe sans cycle**.

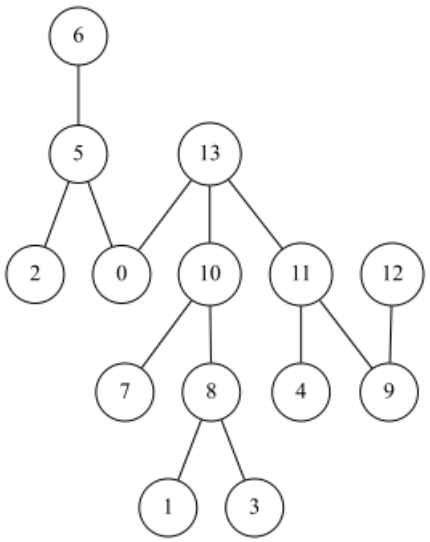
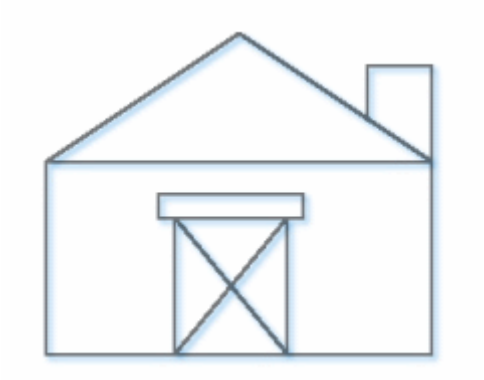


FIGURE 7 – `GT: files/graphISP_tree.gra`

Écrire la fonction `is_tree` qui vérifie si un graphe non orienté simple est un arbre.

Draw me



Exercice 3.6 (Compilation, cuisine...)

1. *Ordonnancement , un exemple simple :*
   Supposons l'ensemble d'instructions suivantes à effectuer par un seul processeur :

   ⓪ lire(a)
   ① b ← a + d
   ② c ← 2 * a
   ③ d ← e + 1
   ④ lire(e)
   ⑤ f ← h + c / e
   ⑥ g ← d * h
   ⑦ h ← e - 5
   ⑧ i ← h - f

   Quels sont les ordres possibles d'exécution ?
   Comment représenter ce problème sous forme de graphe ?

   Chaque solution correspond à un *tri topologique* du graphe.

2. Quelle doit être la propriété du graphe pour qu'une solution de tri topologique existe ?

3. Si on dessine le graphe en alignant les sommets dans l'ordre d'une solution de tri topologique, que peut-on constater ?

4. (a) Soit $suffix$ le tableau contenant les dates de dernière visite : l'ordre suffixe de tous les sommets de G lors d'un parcours en profondeur.
   Démontrer que pour une paire quelconque de sommets distincts $u, v \in S$, s'il existe un arc dans G de u à v, et si G a la propriété de la question 2, alors $suffix[v] < suffix[u]$.
   (b) En déduire un algorithme qui trouve une solution de tri topologique pour un graphe (on supposera qu'une solution existe).
   (c) Que faut-il changer à cet algorithme pour vérifier l'existence d'une solution dans un graphe quelconque ?
   (d) Écrire la fonction Python qui retourne une solution de tri topologique sous la forme d'une liste de sommets si elle existe. La fonction déclenche une exception dans le cas contraire.

Et la cuisine dans tout ça ?