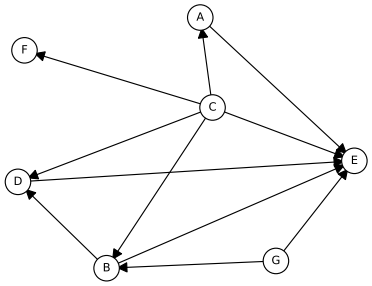
Tri topologique d'un graphe

Contexte

Ce problème étudie les tri topologiques sur les graphes orientés acycliques, c'est à dire les numérotations "croissantes" des sommets d'un tel graphe. Deux algorithmes sont proposés: un algorithme récursif par retrait de sources, et un algorithme de parcours en profondeur.



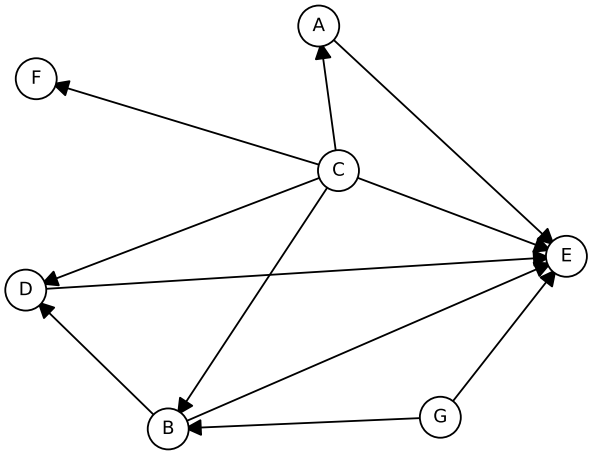
i	C	G	A	F	B	D	E
$t(i)$	0	1	2	3	4	5	6

1 Tri topologique dans un graphe orienté

On considère dans cet exercice des graphes **orientés** $G = (S, A)$ où S est l'ensemble des sommets et A l'ensemble des arêtes.
Un graphe sera représenté par une **dictionnaire d'adjacence** D de taille $|S|$ telle que pour tout sommet $i \in S$, D_i est la liste des sommets $j \in S$ tels que l'arête (i, j) appartient à A.

Par exemple le graphe G_0 à droite est représenté par la dictionnaire d'adjacence suivant:

```
D0 = {
    "A": ["E"],
    "B": ["D", "E"],
    "C": ["B", "A", "D", "E", "F"],
    "D": ["E"],
    "E": [],
    "F": [],
    "G": ["B", "E"],
}
```



On rappelle qu'un chemin de i vers j est une suite de sommets $i = i_0, i_1, \ldots, i_k = j$ avec $k \geq 0$ tels que pour tout $l \in [\![0, k-1]\!]$, l'arête (i_l, i_{l+1}) appartient à A.
Un *cycle* est un chemin de la forme $i = i_0, i_1, \ldots, i_k = i$ avec $k \geq 1$.
Remarque: en particulier on considère q'un chemin de longueur 0 *de* i *vers* i *est un chemin, mais pas un cycle.*
On dit qu'un graphe orienté $G = (S, A)$ est **acyclique** si et seulement si il n'admet pas de cycle.

Pour tous $i, j \in S$, on note $i \prec j$ si et seulement si il existe un chemin de i vers j.
On constate (il n'est pas demandé de le redémontrer) que $\prec$ est une relation réflexive sur S.

1.1. Montrer que $\prec$ est une relation est transitive sur S.
À quelle condition sur G la relation $\prec$ est-elle une relation d'ordre sur S ?

1.2. On considère la fonction python suivante:

```
1  def mystere(D, i):
2      """
3      Fonction mystère et donc sans documentation.
4      """
5      etats = {s: "non_visité" for s in D}
6      liste = []
7
8      def dfs(u):
9          etats[u] = "à_visiter"
10         print(u)
11         for v in D[u]:
12             if etats[v] == "non_visité":
13                 dfs(v)
14         etats[u] = "visité"
```

```
15
16      dfs(i)
17      return None
```

À quel type d'algorithme correspond cette fonction ?
Qu'affiche à l'écran `mystere(D0,"C")` où `D0` est le dictionnaire d'adjacence du graphe orienté G_0 précédent ?

1.3. Écrire une fonction python `est_acyclique` prenant en argument un dictionnaire d'adjacence D représentant un graphe orienté, et renvoyant `True` si le graphe est acyclique et `False` sinon.
Remarque: ne pas oublier de tester toutes les composantes connexes.

On appelle tri-topologique d'un graphe orienté acyclique $G = (S, A)$ une fonction $t : S \to [\![0, |S| - 1]\!]$ telle que

$$\forall i, j \in S,\ i \prec j \Rightarrow t(i) < t(j).$$

Un ordre topologique sera représenté par une liste T de taille $|S|$ telle que pour tout sommet $i \in S$, on a $T[i] = t(i)$.
Par exemple pour le graphe G_0 précédent on a comme tri topologique possible

i	C	G	A	F	B	D	E
$t(i)$	0	1	2	3	4	5	6

1.4. Proposer un tri-topologique du graphe orienté acyclique représenté par la dictionnaire d'adjacence suivante:

```
D1 = {"A": ["B", "C", "D", "E"], "B": ["D"], "C": ["D", "E"], "D": ["E"], "E": []}
```

Remarque: un tableau suffisamment clair ainsi qu'un schéma du graphe pourront faire office de preuve.

Un premier algorithme récursif pour déterminer un tri-topologique d'un graphe orienté acyclique $G = (S, A)$ est le suivant:

- On déterminer un sommet $i \in S$ tel qu'il n'existe pas d'arête (j, i) pour $j \in S$.
- On numérote en premier le sommet i.
- On supprime le sommet i et toutes les arêtes sortantes de i du graphe.
- On recommence jusqu'à ce qu'il n'y ait plus de sommet dans le graphe.

1.5. Écrire une fonction python `liste_sources` prenant en argument un dictionnaire d'adjacence D et renvoyant la liste des sommets $i \in S$ tels qu'il n'existe pas d'arête (j, i) pour $j \in S$.

1.6. Écrire une fonction python `retirer_sommet` prenant en argument un dictionnaire d'adjacence D et un sommet $i \in S$, et renvoyant la dictionnaire d'adjacence du graphe obtenu en retirant le sommet i et toutes les arêtes sortantes de i du graphe représenté par D.
Remarque: on prendra soin de ne pas modifier la dictionnaire d'adjacence D passée en argument.

1.7. Écrire une fonction python `tri_topologique` prenant en argument un dictionnaire d'adjacence D représentant un graphe orienté acyclique, et renvoyant un tri-topologique du graphe représenté par D en appliquant l'algorithme précédent.

Un autre algorithme est le suivant:

- On commence avec une liste vide T représentant l'ordre topologique.
- Tant qu'il reste des sommets non traités
- On choisit un sommet non traité
- On effectue un parcours en profondeur en partant de ce sommet
- On stocke les sommets rencontrés dans l'ordre de leur découverte.
- On rajoute cette liste renversée à la liste T.

1.8. Écrire une fonction python `tri_topologique_dfs` prenant en argument un dictionnaire d'adjacence D représentant un graphe orienté acyclique, et renvoyant un tri-topologique du graphe représenté par D en utilisant l'algorithme de parcours en profondeur.