Tri topologique

1.1

Soient i, j, k \in S$telsquei \prec j$etj \prec k$.

Soit $i = i_0, i_1, \ldots, i_n = j$ un chemin de i à j, et $j = j_0, j_1, \ldots, j_m = k$ un chemin de j à k.

Posons $\ell_0 = i_0, \ell_1 = i_1, \ldots, \ell_n = i_n = j_0, \ell_{n+1} = j_1, \ldots, \ell_{n+m} = j_m = k$.

Alors on a bien

- $\ell_0 = i$, $\ell_{n+m} = k$,
- pour tout $p \in [0, n+m]$, $\ell_p \in S$ et par disjonction de cas (selon que $p \leq n$ ou $p > n$) on a $(\ell_p, \ell_{p+1}) \in A$.

On a donc un chemin de i à k.

Donc $\boxed{\prec \text{ est transitive.}}$

Ainsi $\prec$ est une relation d'ordre sur S **ssi** elle est antisymétrique (car on sait déjà qu'elle est réflexive et transitive).

Soit $i, j \in S$. On a $i \prec j$ et $j \prec i$ **ssi** il existe un chemin de i à j, et un chemin de j à i, c'est à dire un cycle de i à i passant par j.

Ainsi $\prec$ est antisymétrique **ssi** pour tous $i, j \in S$ tel que $i \neq j$, il n'existe pas de cycle passant par i et j.

Donc $\prec$ est une relation d'ordre sur S **ssi** $\boxed{\text{le graphe est acyclique}}$.

1.2

La fonction `mystere` corresponte à un $\boxed{\text{parcours en profondeur}}$ du graphe orienté en partant du sommet i.

En partant de C dans le graphe représenté par `D0` on obtient alors

```
C
B
D
E
A
F
```

1.3

On adapte le parcours en profondeur précédent pour qu'il renvoie faux dès qu'il rencontre un sommet déjà visité.

Et on effectue un parcours en profondeur pour tous les sommets du graphe non déjà visités.

On renvoie dans la récursion du parcours en profondeur `False` dès qu'on rencontre un sommet déjà visité ou si la récursion renvoie `False` .

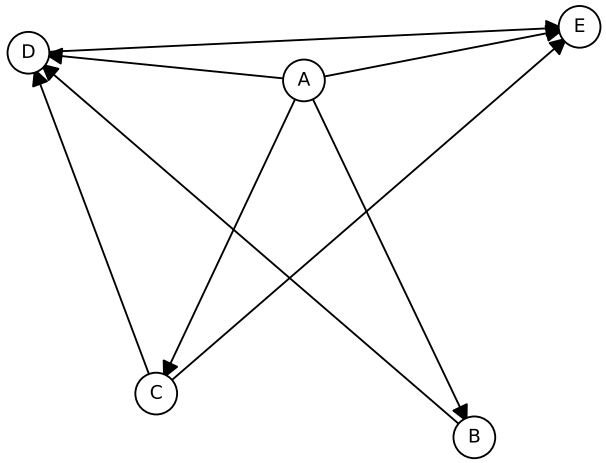
In [1]:
```python
def est_acyclique(D):
    etats = {s: "non_visité" for s in D}

    def dfs(u):
        etats[u] = "à_visiter"
        for v in D[u]:
            # Si on détecte un cycle ou si la récursion a détecté un cylcle
            if (etats[v] == "à_visiter") or (etats[v] == "non_visité" and not dfs(v)):
                return False
        etats[u] = "visité"
        return True

    # On parcourt toutes les composantes connexes
    for sommet in D:
        if etats[sommet] == "non_visité":
            if not dfs(sommet):
                return False
    return True
```

1.4

On obtient le graphe suivant:



et un ordre topologique possible est

i	A	B	C	D	E
t(i)	0	1	2	3	4

1.5

On calcule le degré entrant de chaque sommet, puis on renvoie ceux de degré entrant nul.

In [2]:
```
def liste_sources(D):
    degre_entrant = {s: 0 for s in D}
    for i in D:
        for j in D[i]:
            degre_entrant[j] += 1
    sources = []
    for i in D:
        if degre_entrant[i] == 0:
            sources.append(i)
    return sources
```

1.6

Pour retirer le sommet on parcourt tous les autres sommets et on rajoute les arêtes qui ne vont pas vers le sommet retiré.

In [3]:
```
def retirer_sommet(D, i):
    Dprime = dict()
    for j in D:
        if j != i:
            Dprime[j] = []
            for k in D[j]:
                if k != i:
                    Dprime[j].append(k)
    return Dprime
```

1.7

On effectue alors l'algorithme précédent.

In [4]:
```
def tri_topologique(D):
    T = []
    # Le D.copy() risque de faire une copie superficielle
    # Mais d'un point de vue barème n'est pas pénalisé
    Dmodif = {s: D[s].copy() for s in D}
    while Dmodif:  # Tant qu'il reste des sommets
        s = liste_sources(Dmodif)[0]  # On prend une des souces
        T.append(s)
        Dmodif = retirer_sommet(Dmodif, s)
    return T
```

1.8

On effectue un parcours en profondeur du graphe, et on ajoute les sommets à la fin de la liste de sortie en prenant soin de bien renverser la liste.

In [5]:

```python
from collections import deque


def tri_topologique_dfs(D):
    etats = {s: "non_visité" for s in D}
    T = deque()

    def dfs(u):
        etats[u] = "à_visiter"
        for v in D[u]:
            if etats[v] == "non_visité":
                dfs(v)
        etats[u] = "visité"
        T.appendleft(u)

    for sommet in D:
        if etats[sommet] == "non_visité":
            dfs(sommet)
    return list(T)
```