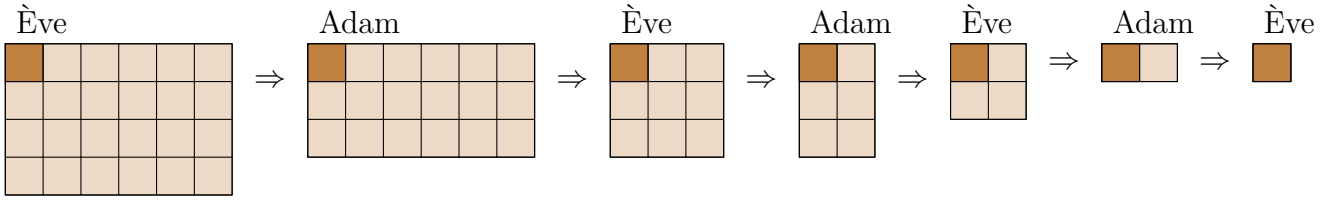
Informatique du Tronc Commun
TD n°5 Théorie des jeux – *Corrigé*

Exercice n°1 Jeu du chocolat empoisonné

On dispose d'une plaquette de chocolat de dimensions $m \times n$. Le carré de chocolat dans le coin en haut à gauche est empoisonné. Chaque joueur à son tour coupe suivant une coupe verticale ou horizontale une part de la plaquette de chocolat qu'il doit manger. Le joueur qui mange le carré empoisonné a perdu.
Voici l'exemple d'une partie avec une tablette de dimensions 4×6. Les joueurs se nomment Adam et Eve. C'est Ève qui commence :

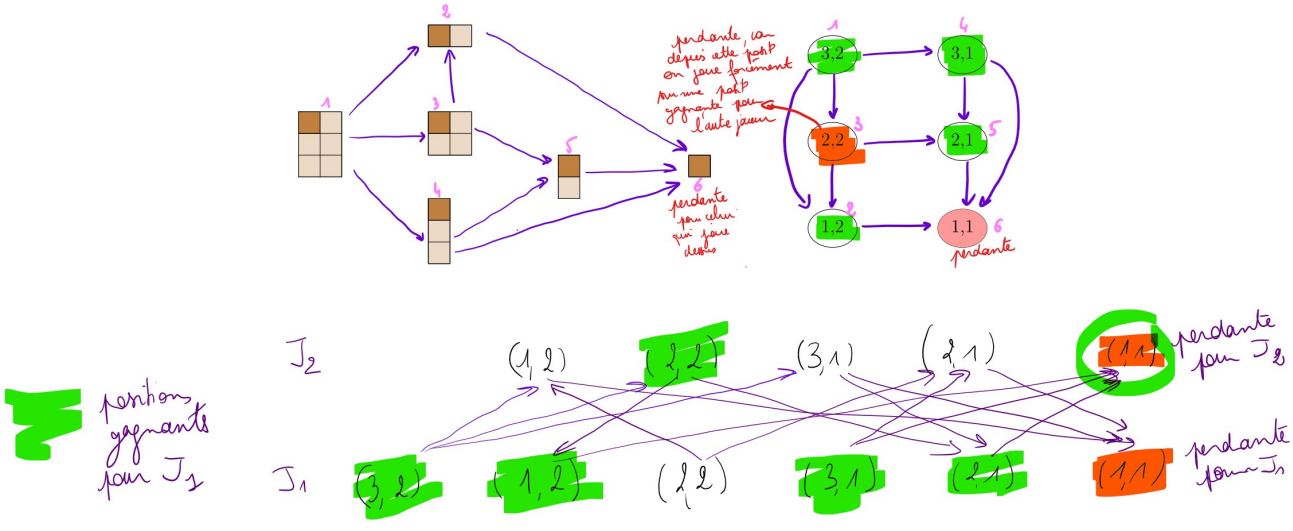


À ce jeu nous pouvons associer un graphe orienté (S, A) appelé Arène du jeu. Les sommets du graphe sont les états possibles de la plaque de chocolat, ici on peut décrire un état par un couple (a, b) avec $0 < a \leq m$ et $0 < b \leq n$.
Un arc relie le sommet s_1 au sommet s_2 lorsqu'un joueur peut, en un coup, passer de l'état s_1 à l'état s_2.

R1. Compléter l'arène du jeu ci-dessous en indiquant les flèches permettant de passer d'un sommet à un autre.

R2. Compléter le graphe du jeu (ci-dessus à droite) pour une tablette initiale 3×2, en indiquant les arêtes.

Solution:



R3. Que peut-on dire la position (1,1) pour le joueur qui doit jouer (gagnante ou perdante ?) ?
Que peut-on en déduire sur les positions (1,2), (2,1), (3,1) (gagnante ou perdante ?) ? En déduire que la position (2,2) est une position perdante. Conclure sur la nature de la position (3,2) ?

R4. Pour la tablette 3×2, que peut-on dire du joueur qui commence ?

Solution: Celui qui commence est en position gagnante, quoique fasse l'autre il gagne.

Exercice n°2 Jeu du chocolat empoisonné (2) : détermination du noyau

Considérons un graphe orienté acyclique (S,A) associé à un jeu d'accessibilité.
Propriété : Dans un graphe acyclique il existe des sommets sans successeur.
Définitions :

— Un sous-ensemble S' de sommets est dit **stable** si tout sommet de S' n'a aucun successeur dans S'.

— Un sous-ensemble S' de sommets est dit **absorbant** si tout sommet n'appartenant pas à S' possède au moins un successeur dans S'.

— Un sous-ensemble S' de sommets est un **noyau** s'il est à la fois stable et absorbant.

Théorème : Tout graphe orienté et acyclique possède un unique noyau.
Pour calculer le noyau de (S, A) il suffit de :

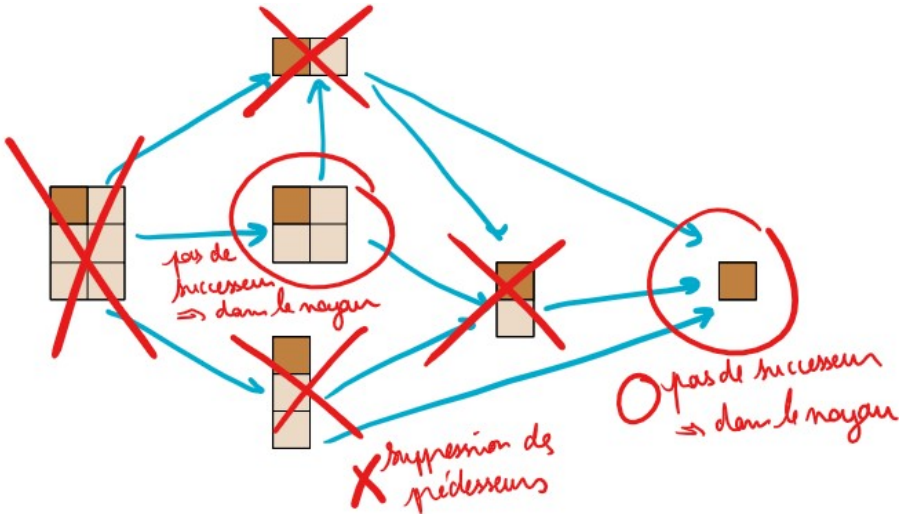
— chercher un sommet s de (S, A) sans successeur, il fait partie du noyau ;

— supprimer de (S, A), s et ses prédécesseurs ;

— recommencer tant qu'il reste des sommets.

R1. Mettre en œuvre l'algorithme précédent pour calculer le noyau du jeu de Chomp (2,3) étudié dans l'exercice précédent.

Solution:



Remarque. Dans le cas du jeu de Chomp les éléments du noyau sont les positions perdantes et les autres sommets les positions gagnantes. La stratégie gagnante consiste, pour chaque sommet qui n'est pas dans le noyau, à jouer un coup qui s'y ramène.

On considère un graphe acyclique (S, A). Il est représenté en machine par la donnée d'un dictionnaire `d` dont les clefs sont les sommets et les valeurs les successeurs des clefs (sous forme de liste).

R2. Rédiger une fonction `sansSucesseur(d:dict)->int` qui renvoie un sommet sans successeur d'un graphe représenté par son dictionnaire `d`.

Solution:

```
def sansSuccesseur(d):
    for c in d: # parcours des clés
        if len(d[c])==0:
            return c
```

R3. Rédiger une fonction `predecesseurs(d:dict,j:int)->list` qui renvoie la liste de tous les prédécesseurs du sommet `j`.

Solution:

```
def predecesseurs(d,j):
    prec_j=[] liste des prédécesseurs de j
    for c in d: # parcours des clés du dictionnaires
```

```
        for x in d[c]: # parcours des successeurs de c
            if x==j: # j est un successeur de c
                prec_j.append(c) # c est un prédécesseur de j
    return prec_j
```

R4. Rédiger une fonction `supprimeSommet(d:dict,j:int)->dict` qui supprime le sommet `j` du graphe représenté par `d`. La fonction modifie le dictionnaire `d`.

Solution:

```
def supprimeSommet2(d,j):
    del d[j]
    for c in d:
        for i in range(len(d[c])): # suppression de j comme successeurs
            if d[c][i]==j:
                del d[c][i]
                break

def supprimeSommet(d,j):
    del d[j] # suppression de la clé de j
    for c in d:
        i=0
        while i<len(d[c]):
            if d[c][i]==j:
                del d[c][i]
            else:
                i=i+1
```

R5. En déduire une fonction `noyau(d:dict)->list` qui renvoie la liste des sommets constituant le noyau de (S, A).

Solution:

```
def noyau(d):
    N=[]
    while len(d)>0:
        s=sansSuccesseur(d)
        N.append(s)
        pred=predecesseurs(d,s) # liste des précédesseurs de j
        supprimeSommet(d,s)
        for x in pred:
            supprimeSommet(d,x)
    return N
```

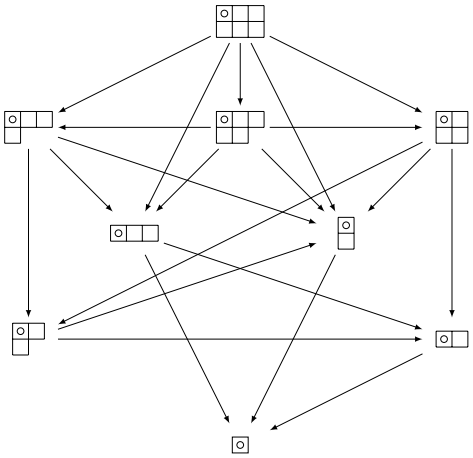
Exercice n°3 Jeu du chocolat empoisonné (3) : Fonction de Sprague-Grundy

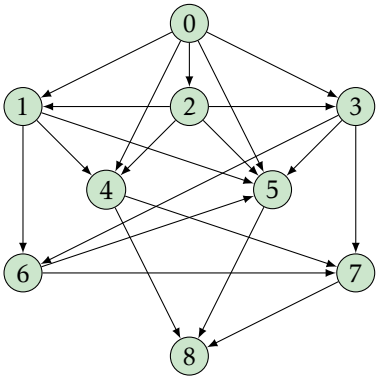
On considère un graphe orienté acyclique (S, A) associé à un jeu impartial dans lequel le perdant est celui qui ne peut plus jouer. On définit le nombre de Grundy, $n(s)$ d'un sommet $s \in S$ de la façon suivante :

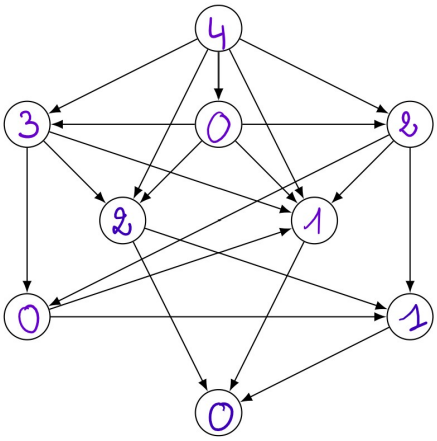
— si s est une position sans successeur (position finale), $n(s) = 0$;

— sinon, le nombre de Grundy $n(s)$ est le plus petit entier positif ou nul qui n'est pas le nombre de Grundy d'un des successeurs de s (c'est-à-dire les positions directement accessibles depuis s).

Le noyau du graphe correspond aux sommets s vérifiant $n(s) = 0$.
On considère le jeu de Chomp (2,4) pour lequel un joueur peut manger à chaque tour un rectangle (qui peut être composé de plusieurs carreaux) dont on donne l'arène de jeu ci-dessous.







R1. Reporter sur le graphe ci-dessus à droite le nombre de Grundy de chacun de ses sommets.

Solution: On en déduit que le noyau est $\{2, 6, 8\}$

On représente un graphe par un dictionnaire `d` dont les clefs sont les sommets et les valeurs la liste des successeurs de la clef correspondante. Le but de cet exercice est de construire un dictionnaire `n` dont les clefs sont les sommets et les valeurs les nombres de Grundy de ces sommets.

R2. Rédiger une fonction `sans_nb_grundy(d:dict, n:dict)->int` qui renvoie un sommet `s` ne possédant pas encore de nombre de Grundy mais dont tous les successeurs en possèdent. Cette fonction renverra `None` dans le cas où un tel sommet n'existe pas.

Solution:

```
def sans_nb_grundy(d,n):
    for s in d: # parcours des clés du dictionnaire d
        if s not in n: # si s n'est pas dans n, s n'a pas de nombre de grundy
            # on cherche maintenant si les succcesseurs de s ont TOUS un nbre de grundy, donc si len(d[s])==nbre de successeurs de s qui sont dans n
            compt=0 # compte le nombre de successeurs de s qui sont dans n, ie qui ont un nbre de grundy
            for x in d[s]: # parcours les successeurs de s
                if x in n: # on compte le nombre de successeurs qui a un nombre de grundy
                    compt=compt+1
            if compt==len(d[s]): # tous les successeurs ont un nbre de grundy
                return s # on a trouvé un sommet qui n'est pas dans n, mais dont tous les successeurs sont dans n
    return None
```

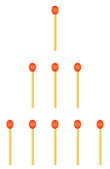
R3. En déduire une fonction `grundy(d:dict)->dict` qui renvoie le dictionnaire `n` des nombres de grundy des différents sommets composants ce graphe représenté par le dictionnaire `d`.

Solution:

```
def grundy(d):
    n={} # dictionnaire des nombres de grundy
    for s in d: # parcours des sommets du graphe
        if len(d[s])==0:# pas de successeurs
            n[s]=0
    while len(n)<len(d):
        s=sans_nb_grundy(d,n) # on récupère un sommet qui n'a pas de nombre de grundy, mais dont les successeurs ont tous un nombre de grundy
        Dn={n[x]:True for x in d[s]} # dictionnaires des nombres de grundy des successeurs (la valeur associée n'a pas d'importance ici)
        nb=0 # on cherche le plus petit entier qui n'est pas déjà attribué à l'un des successeurs
        while nb in Dn: # tant qu'il est dans Dn, nb a déjà été attribué
            nb=nb+1 # on cherche le suivant
        n[s]=nb # on l'a trouvé : on l'attribue
    return n
>>> G={0:[1,2,3,4,5],
   1:[4,5,6],
   2:[1,3,4,5],
   3:[5,6,7],
   4:[7,8],
   5:[8],
   6:[5,7],
   7:[8],
   8:[]}
>>> grundy(G)
{8: 0, 5: 1, 7: 1, 4: 2, 6: 0, 1: 3, 3: 2, 2: 0, 0: 4}
```

Exercice n°4 Jeu de Marienbad

Le jeu de Marienbad est un jeu avec plusieurs tas d'allumettes. Chaque joueur peut retirer autant d'allumettes qu'il veut sur une seule ligne. Le perdant est celui qui retire la dernière allumette.

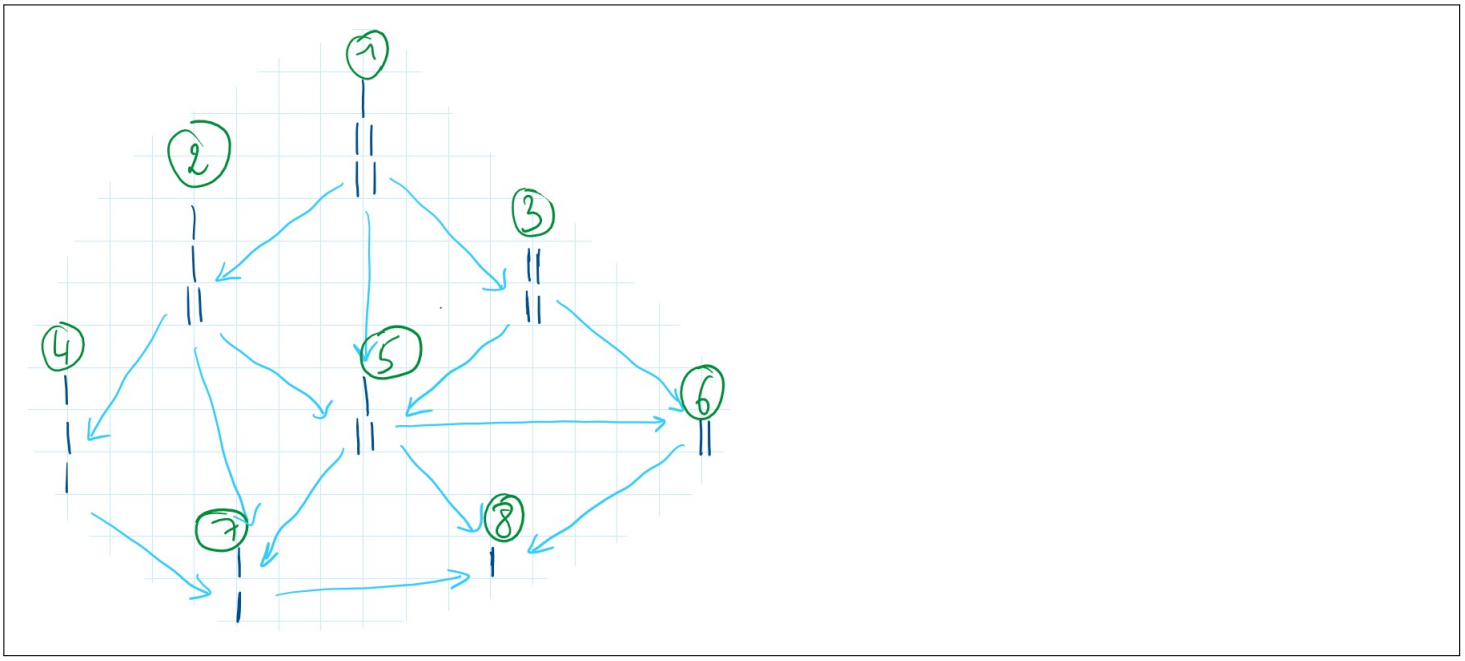


Il est connu pour apparaître dans le film *L'année dernière à Marienbad* d'Alain Resnais (Marienbad est une ville de République Tchèque).

Pour la modélisation par un graphe nous allons considérer une situation plus simple : le jeu a trois lignes dont la première ligne possède une seule allumette et les deux suivantes possèdent deux allumettes | |. Nous considérerons que les situations | | et | | sont identiques (cela permet de limiter le nombre de sommets dans le graphe).

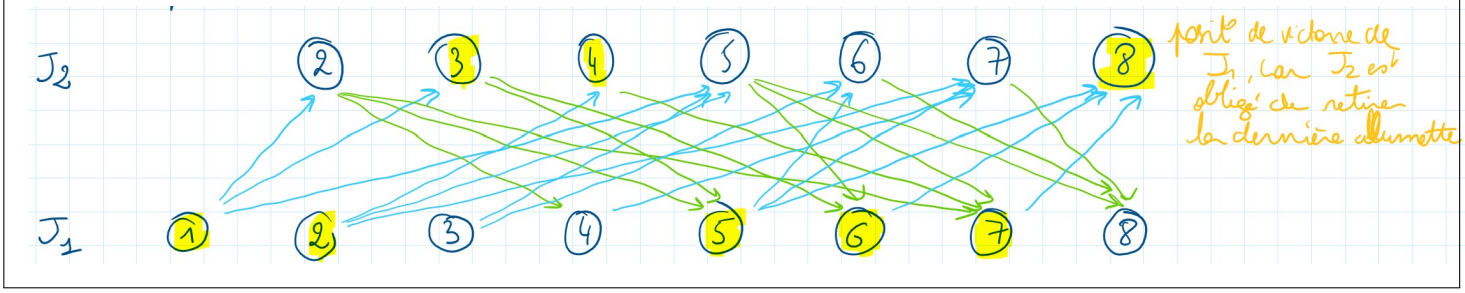
R1. Représenter le graphe du jeu en indiquant les sommets (indice : il y en a 8) et les arcs les reliant (correspondant à des coups qui peuvent être joués).

Solution:



R2. Créer le graphe biparti associé et déterminer l'attracteur du joueur J1 (le premier qui joue).
Vous griserez les états qui sont dans l'attracteur en indiquant le rang de chaque état dans l'attracteur.

Solution:



Modélisation d'un tel jeu

R3. Proposer la modélisation d'un état du jeu (de la question 1) à l'aide d'un tuple.

Solution: On propose un 3-uplet classé par ordre croissant.

```
pos=(t1,t2,t3)
```

`t1,t2,t3` sont le nombre d'allumettes restant dans le tas concerné.

R4. Écrire une fonction `est_trie(t:tuple)->bool` qui prend en argument un tuple `t` qui vérifie si une liste est triée par ordre croissant, elle renvoie un booléen.

Solution:

```
def est_trie(t):
    for i in range(len(t)-1):
        if t[i]>t[i+1]:
            return False
    return True
```

R5. Écrire les instructions qui créent la liste des sommets du graphe de la question 1.

Solution:

```
sommets=[(1,2,2),(1,1,2),(0,2,2),(1,1,1),(0,1,2),(0,0,2),(0,1,1)
   ,(0,0,1)]
```

R6. On suppose avoir écrit `succ(S:list,p:int)->list` qui prend en argument la liste des sommets `S` du graphe ci-dessus, un sommet `p` et qui renvoie la liste de ses successeurs.
Écrire une fonction `graphe(list)->dict` qui prend en argument la liste des sommets `S` et qui renvoie le graphe de la question 1 sous la forme d'un dictionnaire d'adjacence.

Solution:

```
def graphe(S):
    d={}
    for s in S: # parcours des sommets
        x=succ(S,s) # successeurs de s
        d[s]=x # la liste des successeurs est la valeur de la clé s
    return d
```

R7. Écrire une fonction `graphe_biparti(S:list)->dict` qui prend en argument la liste des sommets `S` du graphe ci-dessus et qui renvoie l'arène du jeu (c'est à dire un graphe biparti).

Solution:

```
def graphe_biparti(S):
    arene={}
    G=graphe(S) # dictionnaire du graphe du jeu
    for s in G: # sommets de G
        arene[(1,s)]=[]
        arene[(2,s)]=[]
        for x in G[s]: # successeurs de s
            arene[(1,s)].append((2,x))
            arene[(2,s)].append((1,x))
    return arene
```

On souhaite maintenant calculer l'attracteur et une stratégie gagnante sur ce graphe. On suppose avoir un graphe biparti `G` qui représente l'arène de jeu, V_1 les positions de victoire pour le joueur `J1` et `V2` les positions de victoire pour le joueur `J2`.

R8. Écrire une fonction `nb_succ(G:dict)->dict` qui prend en argument un graphe `G` et qui renvoie un dictionnaire dont les clés sont les sommets de `G` et dont la valeur associée est le nombre de successeurs.

Solution:

```
def nb_succ(G):
    return {s:len(G[s]) for s in G}
```

R9. Écrire un programme `pred(G:dict)->dict` qui prend en argument un graphe `G` et qui renvoie un dictionnaire dont les clés sont les sommets de `G` et dont la valeur associée est la liste des prédécesseurs du sommet.

Solution:

```
def pred(G):
    G2={s:[] for s in G} # dictionnaire des prédécesseurs
    for s in G:
        for x in G[s]:
            G2[x].append(s)
    return G2
```

R10. Compléter le programme récursif `modif_attracteur(s:int,D:dict,TG:dict,S1:list,attr:dict,n:int)-` qui prend en argument un sommet `s`, le dictionnaire du nombre `D` de successeurs qui ne sont pas dans l'attracteur, `TG` le dictionnaire des prédécesseurs de chaque sommet, `S1` les sommets contrôlés par le joueur 1, `attr` un dictionnaire représentant des sommets qui sont dans l'attracteur (non complet a priori) et `n` une majoration du rang du sommet `s` et qui code la partie récursive de l'algorithme.
Ce programme teste si `s` est dans l'attracteur. Si ce n'est pas le cas, il l'ajoute dans l'attracteur et fait les opérations adéquates pour trouver les sommets qui sont dans l'attracteur via un appel récursif.

Solution:

```
def modif_attracteur(s,D,TG,S1,attr,n):
    if s not in attr:
        attr[s]=n # si s n'est pas dans l'attracteur, on l'ajoute
        for pred in TG[s]: # parcours des prédécesseurs de s
            D[pred]=D[pred]-1 # s  successeur de  pred est dans l'
   attracteur, il y a un successeur qui n'est pas dans l'attracteur en
   moins
            if pred in S1 or D[pred]==0: # si pred est contrôlé par S1,
    il fait partie de l'attracteur, ou s'il n'y a pas de successeurs
   qui ne sont pas dans l'attracteur (càd tous les successeurs sont
   dedans), on fait un appel récursif
                modif_attracteur(pred,D,TG,S1,attr,n+1)
```

R11. En déduire une fonction `calcul_attracteur(G:dict,S1:dict,V1:dict)->dict` qui calcule l'attracteur du joueur 1 où `V1` est un dictionnaire des positions de victoire.

Solution:

```
def calcul_attracteur(G,S1,V1):
    d=nb_succ(G) # dictionnaire du nbre de successeurs qui ne sont pas
   dans l'attracteur
    TG=pred(G) # dictionnaire des précédesseurs
    A={} # attracteur
    for s in V1: # on part de la position de victoire
        modif_attracteur(s,d,TG,S1,A,0) # on construit l'attracteur, en
    partant du sommet de V1, au rang  0 (J1 gagne en 0 coup)
    return A
```

R12. Compléter la fonction `condition_gagnanteJ2(G:dict,att1:dict,att2:dict,s:int)->int` qui prend en argument un sommet `s` du graphe `G`, les attracteurs `att1` et `att2` et qui renvoie un sommet optimisé si nous souhaitons gagner la partie pour le joueur 2 (c'est-à-dire une position gagnante si nous sommes dans l'attracteur du joueur 2, ...).

Solution:

```
from random import *

def condition_gagnanteJ2(G,attr1,attr2,s):
    if s in attr2: # J2 gagne s'il joue bien ....
        for x in G[s]:  # parcours des sommets accessibles depuis s
            if x in attr2 and attr2[x]<attr2[s]: # Condition + amélioration du rang (on gagne en moins de coup)
                return x
    elif s in attr1: # J2 a perdu (si J1 joue bien)
        return G[s][randint(0,len(G[s])-1)] #  on joue sur n'importe quel sommet accessible depuis s
    else: # J2 cherche à avoir un match nul
        for x in G[s]: # sommets accessibles depuis s
            if x not in attr1:  # J2  joue sur une position qui n'est pas dans l'attracteur de J1
                return x
```