Chapitre 9

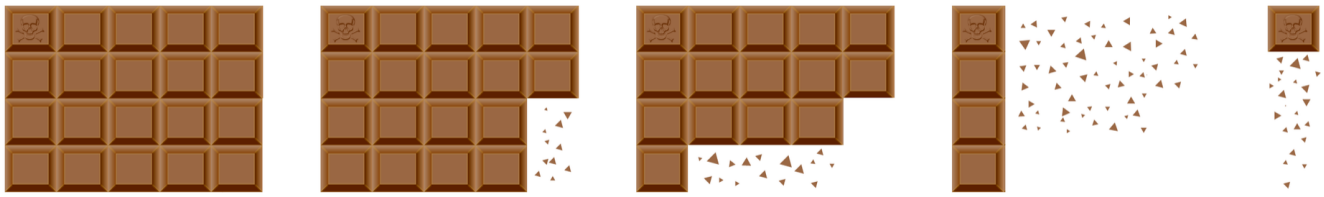
THÉORIE DES JEUX (PARTIE 1)

Dans cette première séance sur la théorie des jeux, le but va être triple :

- *comprendre comment modéliser une partie de jeu à l'aide des* ***graphes bipartis*** *(pour tout jeu tour à tour, fini, à deux joueurs, à information parfaite, et sans hasard, sans match nul) ;*
- *comprendre ce que représente un* ***attracteur****, et comment on les détermine ;*
- *comprendre ce que signifie « avoir une* ***stratégie gagnante*** *» et déterminer les* ***positions gagnantes****.*

Jeu de Nim

Beaucoup des idées présentées ici se servent des parcours de graphes vus en PCSI. Le jeu de Nim constitue un exemple simple et efficace pour appliquer de façon pratique cette théorie. En effet on a déjà trouvé lors de la dernière séance l'ensemble des positions gagnantes ainsi que le joueur qui possédait une stratégie gagnante en fonction du nombre de bâtonnets initiaux. Lors du TP, vous modéliserez le jeu du Chomp, jeu avec des tablettes de chocolat à manger mais qui a également des applications en mathématiques.



Jeu du Chomp

Table des matières

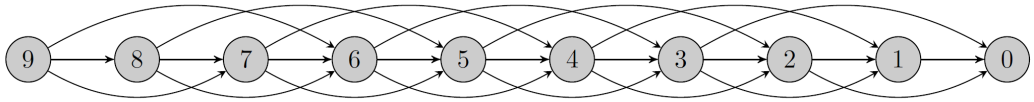
I RAPPELS SUR LE JEU DE NIM

Le **jeu de Nim** (ou des bâtonnets) est un jeu à deux joueurs. Au départ, un tas comporte N bâtonnets, et à son tour un joueur peut en enlever 1, 2 ou 3. Le perdant est celui qui prend le dernier bâtonnet.

On peut modéliser un exemple de partie entre deux joueurs a et b (ici a perd) :

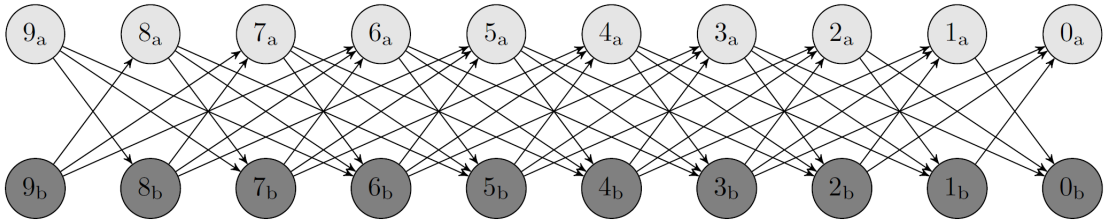
$$20 \xrightarrow{a} 17 \xrightarrow{b} 16 \xrightarrow{a} 15 \xrightarrow{b} 12 \xrightarrow{a} 10 \xrightarrow{b} 8 \xrightarrow{a} 7 \xrightarrow{b} 4 \xrightarrow{a} 3 \xrightarrow{b} 1 \xrightarrow{a} 0$$

Toute partie du jeu de Nim avec $N = 9$ bâtonnets peut se modéliser comme un parcours/chemin sur le graphe orienté ci-dessous.



Un jeu de Nim avec seulement 8 bâtonnets au départ

Le problème sur un tel graphe est qu'il faut noter quel joueur est en train de jouer donc on « dédouble » un graphe comme le précédent en un graphe biparti :



Le graphe biparti associé au jeu de Nim avec 9 bâtonnets au départ.

Remarque. Si le joueur a commence (en 9_a) alors le sommet 9_b ne sera jamais atteint.

Si on veut implémenter ce graphe en Python, on a le choix entre :

- un **couple** `G=(S,A)` où `S` est une liste de sommets et `A` est la liste des arêtes.
- un couple `(D,M)` où `D` est un **dictionnaire** dont les clés sont les numéros de sommets, et la valeur associée est le nom du sommet et `M` la **matrice d'adjacence** : `M[i,j]=1` s'il existe un arc reliant `i` à `j` si le graphe est non valué, 0 sinon.
- `G` un **dictionnaire** dont les clés sont les sommets et la valeur associée à un sommet est la liste de successeurs de ce sommet. Ainsi, `G[s]=[a,b]` signifie que `s` est un sommet et qu'il y a un arc reliant `s` à `a` et un autre reliant `s` à `b`.

C'est cette dernière façon que nous avons modélisé le graphe du jeu de Nim dans le TP précédent.
Construction du graphe du jeu de Nim en itératif :

```
def GrapheNim(N:int,j1:str,j2:str) -> dict:
    '''Renvoie le dictionnaire d'adjacence biparti du jeu de Nim'''
    G={}
    opp={j1:j2,j2:j1}
    for j in [j1,j2]:
        G[(0,j)]=[]
        G[(1,j)]=[(0,opp[j])]
        G[(2,j)]=[(0,opp[j]),(1,opp[j])]
        for k in range(3,N+1):
            G[(k,j)]=[(k-1,opp[j]),(k-2,opp[j]),(k-3,opp[j])]
    return G
```

On peut aussi procéder par récursivité pour créer le graphe si on connait le joueur `j` de départ : en partant d'un sommet de la forme `(n,j)` on visite les trois successeurs de ce sommet : `(n-1,o)`,`(n-2,o)`,`(n-3,o)` où `o` est le nom de l'opposant du joueur, mais en gardant en tête que le nombre de bâtonnets doit être supérieure ou égale à 1.

```
def GrapheNim(N,j):
    """Renvoie le graphe du jeu de Nim avec N bâtonnets et joueur qui commence"""
    if (N,j) not in G: # Si on n'a pas encore rajouté ce sommet
        G[(N,j)]=[(k,Opp[j]) for k in range(max(N-3,1),N)] # Liste successeurs de (N,j)
        for (n,o) in G[(N,j)]:
            GrapheNim(n,o) # On va visiter ce successeur

Opp={"Morgan":"Malo","Malo":"Morgan"} # dictionnaire des opposants
G={} # Initialisation du dictionnaire du graphe modifié par effet de bord
GrapheNim(9,"Morgan")
```

Cette fonction n'est rien de plus qu'un parcours en profondeur récursif. On peut d'ailleurs reprogrammer cette fonction avec un parcours en profondeur avec une pile ou avec un parcours en largeur avec une file :

```
def GrapheNimP(N,j):
    """Graphe jeu de Nim avec N bâtonnets et j qui commence (parcours profondeur)"""
    Pile=[(N,j)] # Pile avec le sommet initial
    while len(Pile)>0: # Tant que toute la pile n'est pas finie
        (N,j)=Pile.pop() # On dépile (le dernier arrivé)
        if (N,j) not in GP: # S'il n'a pas déjà été traitée
            GP[(N,j)]=[(k,Opp[j]) for k in range(max(N-3,1),N)] # Liste successeurs
            Pile=Pile+[(n,o) for (n,o) in GP[(N,j)] if (n,o) not in GP]
            #On rajoute les successeurs à la pile

def GrapheNimL(N,joueur):
    """Graphe jeu de Nim avec N bâtonnets et j qui commence (parcours largeur)"""
    File=[(N,j)] # File avec le sommet initial
    while len(File)>0: # Tant que toute la pile n'est pas finie
        (N,j)=File.pop(0) # On défile (le premier arrivé)
        if (N,j) not in GL:
            GL[(N,j)]=[(k,Opp[j]) for k in range(max(N-3,1),N)] # Liste successeurs
            File=File+[(n,o) for (n,o) in GP[(N,j)] if (n,o) not in GL]

Opp={"Morgan":"Malo","Malo":"Morgan"}

GP={} # Initialisation du dictionnaire du graphe (parcours en profondeur avec pile)
GrapheNimP(9,"Morgan")

GL={} # Initialisation du dictionnaire du graphe (parcours en largeur avec file)
GrapheNimL(9,"Morgan")


print(G==GP and G=GL) # On teste si les 3 graphes obtenus sont identiques.
```

Remarque.

Il n'est pas essentiel de retenir par cœur ce qui précède. L'idée est de comprendre comment représenter les sommets et déterminer les voisins de chaque sommet pour bien représenter le jeu.

Voyons maintenant comment généraliser l'étude des parties faites à la séance précédente, notamment l'étude des stratégies gagnantes, qui passe par celle des positions gagnantes et des attracteurs.

II MODÉLISATION D'UN JEU

On étudie dans ce chapitre des jeux *à deux joueurs, tour à tour, fini, à information parfaite et sans hasard.* Un tel jeu, comme le jeu de Nim, peut se modéliser par un graphe **orienté** **biparti** de sorte qu'une partie dans le jeu se résume simplement à un **chemin** dans le graphe biparti, sans avoir à distinguer quel joueur joue.

Définition.

Un graphe orienté $G = (S, A)$ composé d'un ensemble de sommets S et d'un ensemble d'arcs A est dit **biparti** s'il existe une partition de S en deux sous-ensembles S_1 et S_2 tels qu'aucun arc du graphe ne relie deux sommets de S_1, ou deux sommets de S_2.

Définition.

Une **arène** est un **graphe de jeu** présenté sous la forme de triplet $G = (S_1, S_2, A)$ où (S, A) est un graphe orienté avec S partitionné en S_1 et S_2 (c'est-à-dire $S = S_1 \cup S_2$ et $S_1 \cap S_2 = \emptyset$).

Chaque sommet désigne **une configuration du jeu**. L'un de ces sommets est la **position de départ**.

Les sommets sont parfois appelés états, ou configurations, ou positions.

Remarque.

• Si $s \in S_1$, on dit que c'est un **sommet contrôlé par J_1** : si on est dans ce sommet, c'est à J_1 de jouer (il choisit l'arc du graphe à suivre pour continuer la partie).

• De plus, dans un graphe biparti, les arcs partant d'un sommets de S_1 ne peuvent aller que vers un sommet de S_2 et inversement. Cela traduit qu'une fois que J_1 a joué, c'est à J_2 à jouer. On proscrit donc les jeux où un joueur peut jouer plusieurs fois d'affilée, ou encore ceux où J_1 et J_2 jouent simultanément.

Définition.

Un sommet sans successeur est appelé **sommet terminal** (ou **final**) (le jeu s'arrête donc).

Un sommet sans prédécesseur est appelé **sommet initial**.

On supposera le graphe **fini** et **acyclique** (il n'y a pas de cycle). Ainsi partant d'un sommet initial s_0, on est obligé d'atteindre un sommet terminal en un nombre fini d'étapes, et le jeu s'arrête.

Si l'on note `T` la liste des sommets terminaux, alors $\texttt{T} = \texttt{G1} \cup \texttt{G2} \cup \texttt{N}$. Les sommets de `G1` sont synonymes de victoires pour J_1, les sommets de `G2` sont synonymes de victoires pour J_2 et les sommets de `N` de parties nulles.

Ainsi, une partie où J_1 commence est un chemin de la forme

$$s_0 \xrightarrow[J_1]{} s_1 \xrightarrow[J_2]{} s_2 \xrightarrow[J_1]{} s_3 \longrightarrow \ldots \longrightarrow s_n$$

où $s_i s_{i+1}$ est un arc du graphe. Les sommets s_0, s_2, s_4, ... sont contrôlés par J_1. Au début de la partie, c'est J_1 qui a choisi d'aller s_1 (parmi tous les successeurs de s_0), d'aller en s_3 (parmi les successeurs de s_2)...

En revanche, les sommets s_1, s_3, s_5, ... sont contrôlés par J_2. Cela veut dire que quand on était en s_1 c'est J_2 qui a choisi d'aller en s_2 (parmi tous les successeurs de s_1) etc. Le sommet s_n est un sommet terminal et donc la partie s'arrête. Suivant que s_n appartient à `G1`, à `G2`, ou à `N`, on a le résultat de la partie.

Définition.

Les **jeux d'accessibilité** sont ceux dont le graphe est **biparti** et il y a trois états de fin de partie possibles (victoires de J_1, victoire de J_2 ou nul).

Définition.

Une **stratégie** du joueur J_1 est une fonction $\phi : \left\{ \begin{array}{ccl} S_1 & \to & S_2 \\ s_1 & \mapsto & \phi(s_1) \in G[s_1] \text{ (voisins/successeurs de } s_1) \end{array} \right.$.

Pour J_1, **suivre la stratégie** ϕ signifie que chaque fois que c'est au tour de J_1, si on note s_1 la configuration actuelle, alors J_1 jouera le coup qui mène à la configuration $\phi(s_1)$.

C'est une **stratégie sans mémoire** (les seules à votre programme) : le joueur joue seulement en fonction de la position actuelle s_1 et non en fonction de l'historique de la partie.

Définition.

Une **stratégie** ϕ est dite **gagnante** pour J_1 depuis le sommet s_0 si toute partie jouée depuis s_0 en suivant ϕ à chaque coup de J_1 est gagnante (quel que soit ce que joue J_2).

Une **position** est dite **gagnante** pour J_1 s'il existe une stratégie gagnante partant de cette position.

Propriété (Théorème de Zermelo)**.**

Dans tout jeu tour à tour, fini, à deux joueurs, à information parfaite, et sans hasard, sans match nul, l'un des deux joueurs a une stratégie gagnante.

Le but est maintenant de chercher à construire une stratégie gagnante.

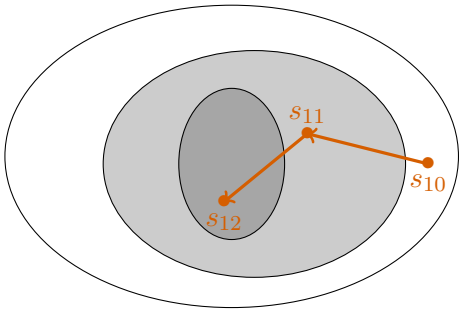
III CALCUL DES ATTRACTEURS

Rappelons que `G1` désigne l'ensemble des sommets terminaux sur lesquels J_1 gagne la partie, on cherche à déterminer comment être sûr d'y arriver, pour cela on va essayer de remonter le temps. Posons :

$$\begin{array}{rrcl} & A_0 & = & \texttt{G1} \\ \forall n \in \mathbb{N} & A_{n+1} & = & A_n \cup \{s \in S_1 \text{ tel que } A_n \cap G[s] \neq \emptyset\} \cup \{s \in S_2 \text{ tel que } G[s] \subset A_n\} \end{array}$$

A_{n+1} est donc l'union :

- de A_n,
- de l'ensemble des sommets contrôlés par J_1 qui permettent d'arriver à A_n (en un coup),
- de l'ensemble des sommets contrôlés par J_2 où quel que soit le coup joué par ce dernier, il sera obligé d'aller dans A_n (en un coup).



On a les ensembles :

- A_0 (en gris foncé),
- A_1 (union du gris foncé et gris clair),
- et A_2 (la grande ellipse).

Supposons par exemple qu'au 10-ième coup, on est au sommet appelé $s_{10} \in A_2$.

— si c'est à J_1 de jouer, il pourra choisir au moins un coup $s_{11} \in A_1$,

— si c'est à J_2 de jouer, tous les coups qu'il pourra faire le mèneront à un sommet dans $s_{11} \in A_1$.

Ensuite, si c'est à J_1 de jouer, il pourra jouer au moins un sommet $s_{12} \in A_0$ et gagner, si c'est à J_2 de jouer, quelque soit le coup qu'il peut jouer, ce sera un coup $s_{12} \in A_0$. Dans tous, les cas, J_1 gagne.

Détaillons ce qui se passe sur les premiers A_n :

- Si on est dans une position qui est dans A_0, alors la partie est gagné par J_1, autrement dit la partie est déjà gagné (en 0 coup).
- Si on est dans une position qui est dans A_1 mais pas dans A_0, alors si c'est au joueur J_1 de jouer, il peut choisir d'aller en un coup dans une position de A_0, si c'est au joueur adverse de jouer, quelque soit le coup qu'il peut jouer, il va tomber dans une position de A_0. Dans les deux cas, J_1 va gagner (en 1 coup).
- si on est dans une position qui est dans A_2 mais pas dans A_1, alors si c'est au joueur de J_1 de jouer, il peut choisir d'aller en A_1, ainsi par ce qui précède il va gagner en deux coups, si c'est au joueur J_2 il va être forcer d'aller en A_1, ainsi J_1 va gagner (en 2 coups).

En généralisant par une récurrence, on peut aboutir à la conclusion suivante : **si on est dans une position de A_n, alors J_1 peut gagner en n coups ou moins**. Ce qui justifie que ces ensembles A_n vont être essentiels pour gagner et qu'il faut être capable de les calculer.

Voici une façon de procéder : soit X un ensemble et J un joueur, alors on note :

$$F(X,J) = (\,\{s \in S_J \text{ tel que } X \cap G[s] \neq \emptyset\} \cup \{s \notin S_J \text{ tel que } G[s] \subset X\}\,) \setminus X$$

De sorte que $A_{n+1} = A_n \cup F(A_n, J_1)$ (en fait F est construit de sorte que $F(A_n, J_1) = A_{n+1} \setminus A_n$).

Pour coder la fonction `F`, on remarque que $X \cap G[s] \neq \emptyset \Leftrightarrow \exists$ successeur $\in G[s]$ t.q. successeur $\in X$, et que $G[s] \subset X \Leftrightarrow \forall$ successeur $\in G[s]$, successeur $\in X$. On va s'aider de deux fonctions auxiliaires `Existence(X,s)` et `PourTout(X,s)`. La première renverra `True` si l'un des successeurs du sommet `s` est dans X, et la seconde renverra `True` si tous les successeurs de `s` sont dans X :

```
def Existence(X,s):
    for succ in G[s]:
        if succ in X:
            return True
    return False
```

```
def PourTout(X,s):
    for succ in G[s]:
        if succ not in X:
            return False
    return True
```

On code alors la fonction `F(X,j)` en bouclant sur tous les sommets du graphe. On suppose que chaque sommet `s` est codé sous la forme d'un couple `s=(position,j)` où `j` est le joueur qui contrôle le sommet. On distingue les cas où le sommet est contrôlé par le joueur `j` ou par son adversaire :

```
def F(X,j):
    L=[]
    for s in G:
        if s not in X and s[1]==j and Existence(X,s): # s est contrôlé par j
            L.append(s)
        if s not in X and s[1]!=j and PourTout(X,s): # s est contrôlé par l'adversaire
            L.append(s)
    return L
```

Ainsi, pour tout $n \in \mathbb{N}$, $A_{n+1} = A_n \cup F(A_n, J_1)$.

En particulier $A_n \subseteq A_{n+1}$, c'est à dire que la suite d'ensembles (A_n) est croissante au sens de l'inclusion.

Comme le graphe est fini, il existe $N \in \mathbb{N}$ tel que $A_N = A_{N+1}$. De plus, pour ce $N \in \mathbb{N}$, on a $A_N = \bigcup_{n\in\mathbb{N}} A_n$.

C'est à dire que la suite d'ensembles (A_n) est stationnaire (*constante à partir d'un certain rang*).

A_N est l'ensemble des **positions gagnantes** de J_1 (positions où J_1 est assuré de pouvoir gagner s'il joue bien).

Définition.

On dit que A_N est **l'attracteur** pour J_1 : si on tombe dans l'attracteur alors on y reste (*à condition que J_1 joue « bien »*) quoique fasse J_2.

```python
def Attracteur(j):
    X=[(1,Opp[j])] # X=A0, liste des sommets gagnants pour le joueur j (ici jeu de Nim)
    Y=F(X,j)
    while len(Y)>0: # Tant qu'il y a des choses à rajouter
        X=X+Y # On actualise X, à la nième itération X=An : X=X+f(X)=X+f(A_{n-1})=A_n
        Y=F(X,j)
    return X

print(Attracteur("Morgan"))
```

Le calcul de l'attracteur pour J_1 donne toutes les positions gagnantes pour J_1.
Le calcul de l'attracteur pour J_2 donne toutes les positions gagnantes pour J_2.
Toutes les autres positions mènent à un match nul (ou à une partie infinie si le graphe de jeu contient des cycles).
En particulier un joueur a une stratégie gagnante pour le jeu si et seulement si la position initiale est dans son attracteur.
S'il n'y a pas de position finale correspondant à un match nul (et que le graphe de jeu ne contient pas de cycles) alors il suffit de calculer l'attracteur d'un seul joueur : toutes les positions qui ne sont pas dans son attracteur sont gagnantes pour l'autre joueur.

IV CONSTRUCTION D'UNE STRATÉGIE GAGNANTE

Avec le calcul d'attracteur fait précédemment, on peut construire une stratégie gagnante en ne parcourant que des sommets qui sont des éléments de A_N.

Définition.

Pour tout sommet s dans l'attracteur (i.e. $s \in A_N$) on définit le **rang** de s par :

$$r(s) = \min\{k \in \mathbb{N} \text{ tel que } s \in A_k\}$$

Soit $s \in A_N$, supposons que ça soit au tour de J_1 de jouer alors qu'il est en position s (en particulier $s \in S_1$).

— Si $r(s) = 0$, alors $s \in A_0 =$ `G1` et J_1 a donc gagné la partie,

— sinon, notons $n = r(s) - 1$, alors $s \in A_{n+1}$ et $s \notin A_n$

Ainsi, parmi les successeurs de s il existe $v \in A_n$. En particulier $r(v) \leqslant n < r(s)$, on pose alors, $\phi(s) = v$ et J_1 joue la position v.

Maintenant, J_2 doit jouer depuis la position v. Si $r(v) = 0$, alors $v \in A_0 =$ `G1` et J_1 a donc gagné la partie. Sinon notons $m = r(v) - 1$, alors $v \in A_{m+1}$ et $v \notin A_m$, ainsi n'importe quel coup c que peut jouer J_2 est dans A_m et vérifie donc $r(c) \leqslant m < r(v)$.

Ainsi, partant d'un sommet de l'attracteur, J_1 peut toujours choisir un coup dont le rang est strictement plus petit, et quelque soit le coup que J_2 joue, il atteindra une position dont le rang est encore strictement plus petit. Comme le rang diminue strictement et reste positif, on arrive nécessairement au bout d'un temps fini au rang 0, c'est à dire à la victoire de J_1.

Il suffit donc de coder le rang des éléments de l'attracteur (= nombre d'itérations à faire pour atteindre la victoire de J_1), puis parmi tous les successeurs d'un élément, en retourner un dont le rang est plus petit. La fonction ϕ ainsi construite est donc une stratégie gagnante si on part d'un point de l'attracteur.

Plus précisément, voici une fonction qui, étant donné un joueur et un sommet dont on sait qu'il est une position gagnante pour le joueur, renvoie un coup à jouer depuis ce sommet pour suivire une stratégie gagnante (basée sur le calcul de l'attracteur et du rang).

```
def Strategie(joueur,sommet):
    n = 0
    X = [(1,Opp[joueur])] # X=A_0, Liste des sommets gagnants pour le joueur
    R = {s:n for s in X} # dictionnaire des rangs, R[s] est le rang de s
    while sommet not in R: # tant que l'attracteur n'atteint pas le sommet voulu
        n = n+1 # on actualise n
        Y = F(X,joueur) # on regarde les nouveaux sommets obtenus
        for s in Y:
            R[s] = n  # on enregistre leur rang
        X = X+Y  # On actualise X, à la nième itération X=An
    for v in G[sommet]: # pour tous les successeurs de sommet
        if v in R and R[v]<R[sommet]:
            return v  # successeur de sommet dans l'attracteur de rang plus petit

Strategie("Malo",(6,"Malo")) # Coup à jouer pour Malo pour se rapprocher de la victoire.
```

Remarque.
Que faire si on n'est pas sur un point de notre attracteur ? Choisir aléatoirement un sommet (qui n'est pas dans l'attracteur de l'adversaire si possible) et espérer que votre adversaire fasse une erreur pour que la suite de la partie tombe dans votre attracteur.

Avez-vous une stratégie gagnante pour gagner aux échecs ?

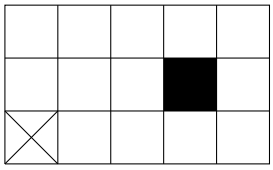
V EXERCICE : JEU DU CHOMP (OU TABLETTE DE CHOCOLAT)

Imaginons une tablette de chocolat avec `n` lignes et `p` colonnes.

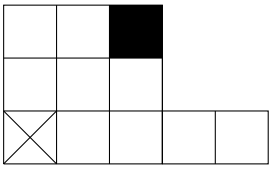
On suppose que le carré situé tout en bas et tout à gauche est empoisonné. Deux joueurs jouent à tour de rôle. À chaque coup, le joueur choisit l'un des carrés de la plaque et mange ce carré ainsi que ceux situé en haut et à droite de ce carré. Le jeu continue tant qu'il reste des carrés à manger. Celui qui a mange le carré empoisonné perd.

Par exemple, partons d'une tablette avec 3 lignes et 5 colonnes.

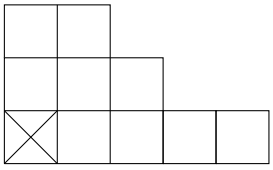
Morgan choisit la case à la 2-ième ligne 4-ième colonne.



Malo choisit alors la case 1ère ligne, 3-ième colonne



La partie continue... tant qu'il reste des carrés à manger...



1. Justifier qu'il s'agit bien d'un jeu d'accessibilité.

   .....

2. Téléchargez le fichier `TP09.py` qui contient des fonctions codant ce jeu.
   Le réglage par défaut de Spyder est d'afficher les fenêtres graphiques dans un onglet « Graphes ». Il faut que cela soit le cas ici (et pas dans une fenêtre indépendante). Donc au préalable de faire le réglage suivant : Outils → Préférences → Console IPython → Graphiques et régler « Sortie graphique » sur « En ligne ».

3. Jouer une partie avec votre voisin (en actualisant la liste `L` avec vos prénoms). Il suffit d'exécuter le code du fichier `TP09.py` et de suivre les instructions.
   Voici quelques commandes utiles sur les ensembles, variable avec laquelle est codée le jeu de Chomp.

   - Les ensembles en Python se comporte comme en maths : pas de répétition, l'ordre ne compte pas.
   - Pour définir l'ensemble vide on écrit `E=set()` (et non `E={}` qui définit un dictionnaire vide.).
   - Pour écrire un ensemble par extension, on fait `E=`$\{1,2,3\}$
   - Par compréhension : `E=`$\{$`k*k for k in range(10) if k!=8`$\}$.
   - À un ensemble `E`, on rajoute un élément par `E.add(8)`.
   - Si `A` et `B` sont deux ensembles `A-B` désigne l'ensemble $A \setminus B$.
   - On peut boucler sur un ensemble comme sur une liste `for e in E:`.
   - `{1,1,3,1,5}=={1,3,5}` est un booléen qui vaut `True`.
     En revanche, `[1,1,3,1,5]==[1,3,5]` est un booléen qui vaut `False`.
   - `len(E)` renvoie le nombre d'éléments (distincts donc) de l'ensemble `E`.

4. Lisez rapidement les fonctions du fichier et comprenez ce qu'elles font.

   `Milka(n,p)` : .....

   `Montrer(tab)` : .....

   `Mange(tab,i,j)` : .....

   `JoueurHumain(tab,joueur)` : .....

   `Partie(n,p,S1,S2,afficher)` : .....

5. Rejouer une partie avec une seule ligne ou une seule colonne mais avec au moins deux carrés de chocolat. Celui qui commence doit pouvoir gagner automatiquement, pourquoi ?

   .....

On souhaite maintenant utiliser l'algorithme de l'attracteur pour construire une autre stratégie. On va construire le graphe du jeu où les sommets seront de la forme `(tab,j)` ou `tab` est l'état de la tablette et `j` désigne le joueur qui doit jouer.

6. Créer une fonction `ListeVoisins(s)` qui renvoie la liste des voisins d'un sommet `s` du graphe `G`.
7. Créer une fonction `GrapheTab(tab,joueur)` récursive (par exemple) qui va créer le graphe `G` par un parcours en profondeur (par exemple). On pourra adapter l'algorithme du cours.
   Créer le graphe `G` pour `n` $= 4$ et `p` $= 3$. Vérifier qu'il contient 66 sommets.

   Warning : Les ensembles ne sont pas hashables.
   Donc ici, comme une position contient un ensemble de carrés de chocolat, elle ne peut être une clé d'un dictionnaire car une variable de type ensemble (comme de type liste) n'est pas hashable.
   À la place, on va juste créer la liste des sommets et on va noter `G` cette liste. À la place de définir la valeur associée à la clef comme la liste des sommets voisins, si on a besoin de la liste des sommets voisins de `s` on utilisera la fonction `ListeVoisins`.
8. Créer une fonction `Attracteur(joueur)` qui calcule l'attracteur (adapter l'algorithme du cours).
   Vérifier que chaque joueur admet 33 sommets attracteurs. Comment savoir lequel des deux joueurs possède une stratégie gagnante ?

   .....

   .....
9. Compléter la fonction `StrategieAleatoire(tab,joueur)` afin qu'elle renvoie une position `s` choisie au hasard dans `tab`, excepté le carré empoisonné (sauf si le joueur `joueur` n'a pas le choix.)
   On pourra utiliser la fonction `np.random.randint(n)` qui renvoie un entier choisi aléatoirement entre 0 et $n-1$.
10. Écrire une fonction `StrategieGagnante(tab,joueur)` qui va utiliser le calcul de l'attracteur de la façon suivante : si la position n'est pas dans l'attracteur renvoyer un coup au hasard, sinon renvoyer un coup qui reste encore dans l'attracteur.
11. Exécuter la dernière cellule de l'algorithme en enlevant les `"""` sur les lignes de code.
    Comprendre le code proposé et expliquer ce qu'il fait.

    .....

    .....