# CORRECTION DU TD ITC$^2$ N° 10: ALGORITHMIQUE POUR L'I.A. 2
## JEUX

## EXERCICE N°1: Analyse d'un graphe biparti

On propose le code suivant

Listing 1:

```
dictio={0:[1,2], 2:[3], 3:[1], 1:[], 4:[1]}
dictiop={0:[1,2], 2:[3], 3:[1], 1:[], 4:[1,0]}

def verif_bipart(dictio:dict):
    V0=[]
    V1=[]
    Bool=True
    for cle in dictio:
        if cle not in V0 and cle not in V1: #si sommet jamais rencontré
            V0.append(cle) #on l'ajoute à V0 (arbitraire)
        if cle in V0: # si sommet dans V0
            for vi in dictio[cle]: #on itère sur ses sommets adjacents
                if vi in V0: #s'il est dans V0 alors NON BIPARTI
                    Bool=False
                if vi not in V1: # si l'adjacent n'est pas dans V1
                    V1.append(vi) # on l'y ajoute

        if cle in V1: #si sommet dans V1
            for vi in dictio[cle]: #on itère sur ses sommets adjacents
                if vi in V1:
                    Bool=False #s'il est dans V1 alors NON BIPARTI
                if vi not in V0: # si l'adjacent n'est pas dans V0
                    V0.append(vi) # on l'y ajoute

    if Bool:
        return V0,V1
    else:
        return False
```

## EXERCICE N°2: Etat gagnant dans le jeu du morpion

Le jeu du Morpion ou *OXO*, consiste pour chaque joueur à tenter de réaliser un **alignement de 3 de ses jetons** sur une grille carrée de 9 cases. Le joueur $J0$ possède les jetons $O$, et le joueur $J1$ les jetons $X$; un état du jeu lors d'une partie est par exemple:

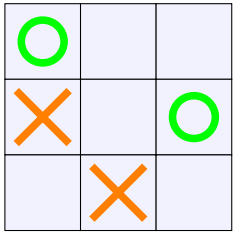

Figure 1: Exemple d'état du jeu au *Morpion*

On supposera que **l'état du jeu** à tout moment est implémenté par un tableau `numpy` dans lequel les jetons $O$ sont des 0 et les jetons $X$ sont des 1, et qu'une case vierge de la grille contient la valeur `infinie` accessible par le module `math` avec la commande `math.inf`; la grille en début de partie sera donc implémentée à l'aide des commandes suivantes:

```
import numpy as np
import math as m
jeu=np.full((3,3), m.inf)
```

❶ Proposer une fonction `grille_pleine(jeu:array) → bool` recevant en entrée le tableau `jeu` représentant la situation de jeu et renvoyant le booléen `True` si la grille est pleine, c'est à dire entièrement remplie de 0 ou de 1, et `False` sinon.

Listing 2:

```
def grille_pleine(jeu}:
        pleine=True
```

```
        for i in range(3):
            for j in range(3):
                if jeu[i,j]==math.inf:
                    pleine=False
        return pleine
```

❷ Proposer une fonction `gagne(jeu:array,J) → bool` recevant en entrée le tableau `jeu` représentant la situation de jeu ainsi que le numéro $J$ d'un des deux joueurs (0 ou 1) et renvoyant le booléen `True` si le joueur $J$ remporte la partie ou `False` sinon.

Listing 3:

```
def gagne(jeu:array,J):
    gagne=False
    for i in range(3):
        if [jeu[i,0],jeu[i,1],jeu[i,2]]==[J,J,J]
            gagne=True
    for j in range(3):
        if [jeu[0,j],jeu[1,j],jeu[2,j]]==[J,J,J]:
            gagne=True
    if [jeu[0,0],jeu[1,1],jeu[2,2]]==[J,J,J] or [jeu[0,2],jeu[1,1],
    jeu[2,0]]==[J,J,J]:
        gagne=True
    return gagne,J
```

❸ Proposer enfin une fonction `Etat_final(jeu:array) → (RES:boolean,J:int)` qui prend en argument le tableau de l'état du jeu `jeu` et renvoie un tuple contenant un booléen `True` ou `False` suivant que la partie est terminée ou non, ainsi qu'un entier $J$ indiquant le joueur ayant remporté la partie: 0 si c'est $J0$, 1 si c'est $J1$, et 2 en cas de partie non terminée ou de match nul.

Listing 4:

```
Etat_final(jeu):
    if gagne(jeu,0)==(True,0):
        return (True,0)
    elif gagne(jeu,1)==(True,1):
        return (True,1)
    elif grille_pleine==True:
        return (True,2)
    else:
        return (False,2)
```

## EXERCICE N°3: Construction de stratégies gagnantes avec l'attracteur pour le jeu de Nim

On reprend ici le cas du jeu de Nim implémenté par un graphe biparti $(V0, V1, A)$ vu en cours pour lequel chaque joueur peut retirer 1,2, ou 3 jetons à chaque coup. On rappelle le corps principal de l'algorithme de calcul de l'attracteur:

Listing 5: Attracteur

```
def attracteur(V,A,V0,F0):
    Attr=[]
    pred,deg={},{}
    for v in V:
        pred[v] = []
        deg[v] = 0
    for a in A:
        p,q = tuple(a)
        deg[p] += 1
        pred[q].append(p)
    def parcours_inv(v):
        if v not in Attr:
            Attr.append(v)
            for u in pred[v]:
                deg[u]-=1
                if u in V0 or deg[u]==0:
                    parcours_inv(u)
        return None
    for x in F0:
        parcours_inv(x)
    return Attr
```

avec $V0$ les sommets contrôlés par $J0$, et $F0$ les états gagnants pour le joueur $J0$.

On souhaite modifier le code afin qu'il renvoie pour chaque position gagnante apparaissant dans l'attracteur la stratégie gagnante correspondante, c'est à dire la liste des sommets à emprunter depuis cette position gagnante pour rejoindre un état gagnant, et ce quels que soient les coups joués par le joueur adverse.

❶ Quelle structure de données vous semble bien adaptée désormais pour l'attracteur (c'est une liste dans le code vu en cours)? Proposer une implémentation.

On peut implémenter l'attracteur sous forme d'un dictionnaire recevant comme clés les différentes positions gagnantes, et comme valeurs, les sommets à emprunter depuis cette position gagnante pour atteindre l'état gagnant.

❷ Proposer les modifications nécessaire du code ci-dessus.

Listing 6:

```
def attracteurs(V,A,V0,F0):
    Attr,pred,deg={},{},{}
    for v in V:
        pred[v] = []
        deg[v] = 0
    for a in A:
        p,q = tuple(a)
        deg[p] += 1
        pred[q].append(p)

    def parcours_inv(v,L): #procédure de parcours inverse à partir du sommet v
        if v not in Attr: #si v n'est pas dans l'attracteur
            L=[v]+L # on ajoute le sommet v à la stratégie gagnante (attention à l'ordre)
            Attr[v]=L #on ajoute en valeur du sommet v les pos. gagn.
            for u in pred[v]: #on itère sur tous les prédécesseurs du sommet v
                deg[u]-=1 #et pour chacun, on retire un degré sortant puisque traité!
                if u in V0 or deg[u]==0: #si u contrôlé par J0 ou est de degré sortant nul
                    parcours_inv(u,L)#alors on relance la procédure de parcours avec ce nouveau sommet
        return None
    for x in F0: #pour tous les sommets dans $F0$:
        parcours_inv(x,[]) #on lance le parcours inverse du graphe
    return Attr
```

## EXERCICE N°4: Construction d'une stratégie gagnante par algorithme minmax

On souhaite modifier l'algorithme du minmax vu en cours afin qu'il permette, outre l'affichage de l'étiquette du vainqueur: $+1$ si $J_0$ remporte la partie, et $-1$ si c'est $J_1$, celui de la liste des sommets empruntés par le vainqueur dans l'arbre de la partie, c'est-à-dire la stratégie gagnante.

On rappelle le code Python de l'algorithme du `minmax` vu en cours:

Listing 7: Algorithme minmax

```
def minmax(sommet,arbre):
        if arbre[sommet]==[]:
                if ('max' in sommet):
                        return +1
                else:
                        return -1
        else: #sinon on lance la récursion minmax
                if ('max' in sommet):
                        return max([minmax(fils, arbre) for fils in arbre[sommet]])
                else:
                        return min([minmax(fils,arbre) for fils in arbre[sommet]])
```

1. Proposer, mais sans le coder, le principe d'une méthode qui permettra de constituer et d'afficher la suite des sommets de l'arbre du jeu empruntés par les joueurs lorsqu'ils suivent l'algorithme du `minmax`.

   Le principe va consister à parcourir chacune des branches de l'arbre depuis la racine et à appliquer l'algorithme du Minmax à chaque fils.

   - on parcourt chacune des branches de l'arbre depuis la racine et on applique l'algorithme du Minmax à chaque fils (max si le fils est contrôlé par J0, min sinon).
   - une liste de sommets empruntés est initialisée à chaque fils et sera remplie par récursion de l'algorithme MinMax
   - enfin, on prendra le maximum de cette liste si le sommet de chaque récursion était max, et le minimum s'il était min.

2. Compléter le code suivant pour qu'il affiche l'étiquette du vainqueur et la liste des sommets empruntés:

   **Attention:** il y a une subtilité dans ce code avec la copie de la liste de départ

Listing 8: Algorithme minmax modifié

```python
import copy as cp

def minmax(sommet,arbre,L):
    if arbre[sommet]==[]:
        if ('max' in sommet):
            return (+1,L)
        else:
            return (-1,L)
    else:
        if ('max' in sommet):
            liste=[]
            for fils in arbre[sommet]:
                L.append(fils) #on ajoute fils à L
                L1=cp.deepcopy(L) #copie conforme de L
                liste.append(minmax(fils, arbre,L1)) # on établit la liste des sommets empruntés depuis fils
                L.pop() # on élimine le sommet fils et sa descendance complète déjà explorée
            return max(liste)
        else:
            liste=[]
            for fils in arbre[sommet]:
                L.append(fils)
                L2=cp.deepcopy(L)
                liste.append(minmax(fils,arbre,L2))
                L.pop()
            return min(liste)
```