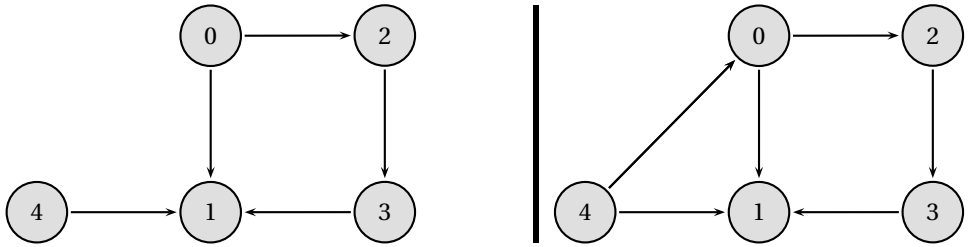
TD ITC^2 N° 9: ALGORITHMIQUE POUR L'I.A. 2
JEUX

EXERCICE N°1: Analyse d'un graphe biparti

On reprend ici l'exemple élémentaire du cours présentant deux graphes: l'un biparti, l'autre non.



Proposer un code python `vérif_biparti(dictio_graphe:dict)` recevant en argument le dictionnaire d'adjacence d'un graphe, et renvoyant les deux sous-ensembles de sommets V_0 et V_1 si ce dernier est biparti, et `False` sinon.

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:

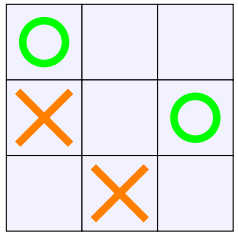


Figure 1: Exemple d'état du jeu au *Morpion*

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.

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.

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.

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 1: Attracteur

```python
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)
    L=[]
    def parcours_inv(v,L):
        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.

❷ Proposer les modifications nécessaire du code ci-dessus.

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 $J1$), 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 2: Algorithme minmax

```python
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`.

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

Listing 3: 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(.....)
                L1=.....
                liste.append(minmax(fils, arbre,L1))
                .....
            return max(.....)
        else:
            liste=[]
            for fils in arbre[sommet]:
                .....
                L2=.....
                liste.append(minmax(fils,arbre,.....))
                .....
            return min(.....)
```