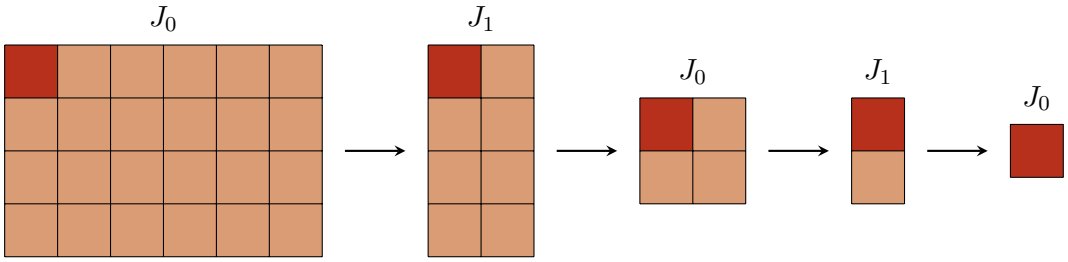
TP9 - Théorie des jeux

I. Étude d'un jeu

Deux joueurs J_0 et J_1 jouent avec une tablette en chocolat composée de p lignes et q colonnes de carreaux. Chaque joueur choisit à tour de rôle soit de manger une partie des dernières lignes en bas de la tablette, soit de manger une partie des dernières colonnes à droite de la tablette. Le carreau du coin supérieur gauche est empoisonné : le joueur qui se trouve dans l'obligation de le manger perd la partie. Le joueur J_0 est le premier à jouer.

Voici une partie avec une tablette de taille $(p, q) = (4, 6)$ dans laquelle le joueur J_1 gagne.

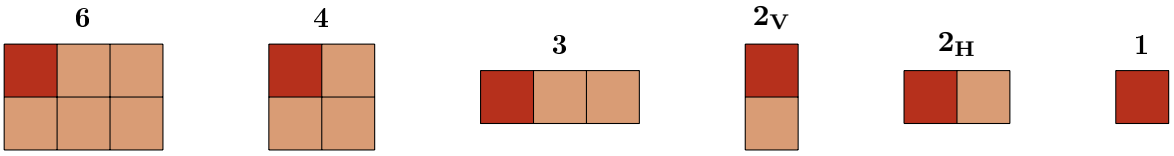


I.1. Étude d'un cas particulier

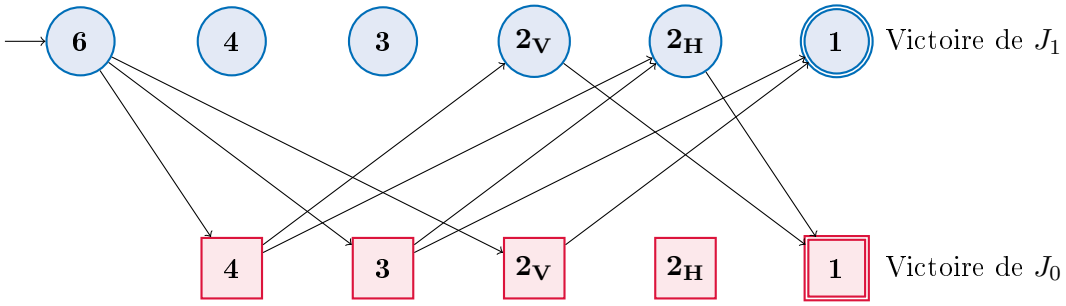
Dans cette sous-partie, on suppose : $(p, q) = (2, 3)$.

► Représenter le graphe biparti associé à ce jeu.

Tout d'abord, nommons les différentes configurations possibles (en fonction de leur nombre de cases par exemple).

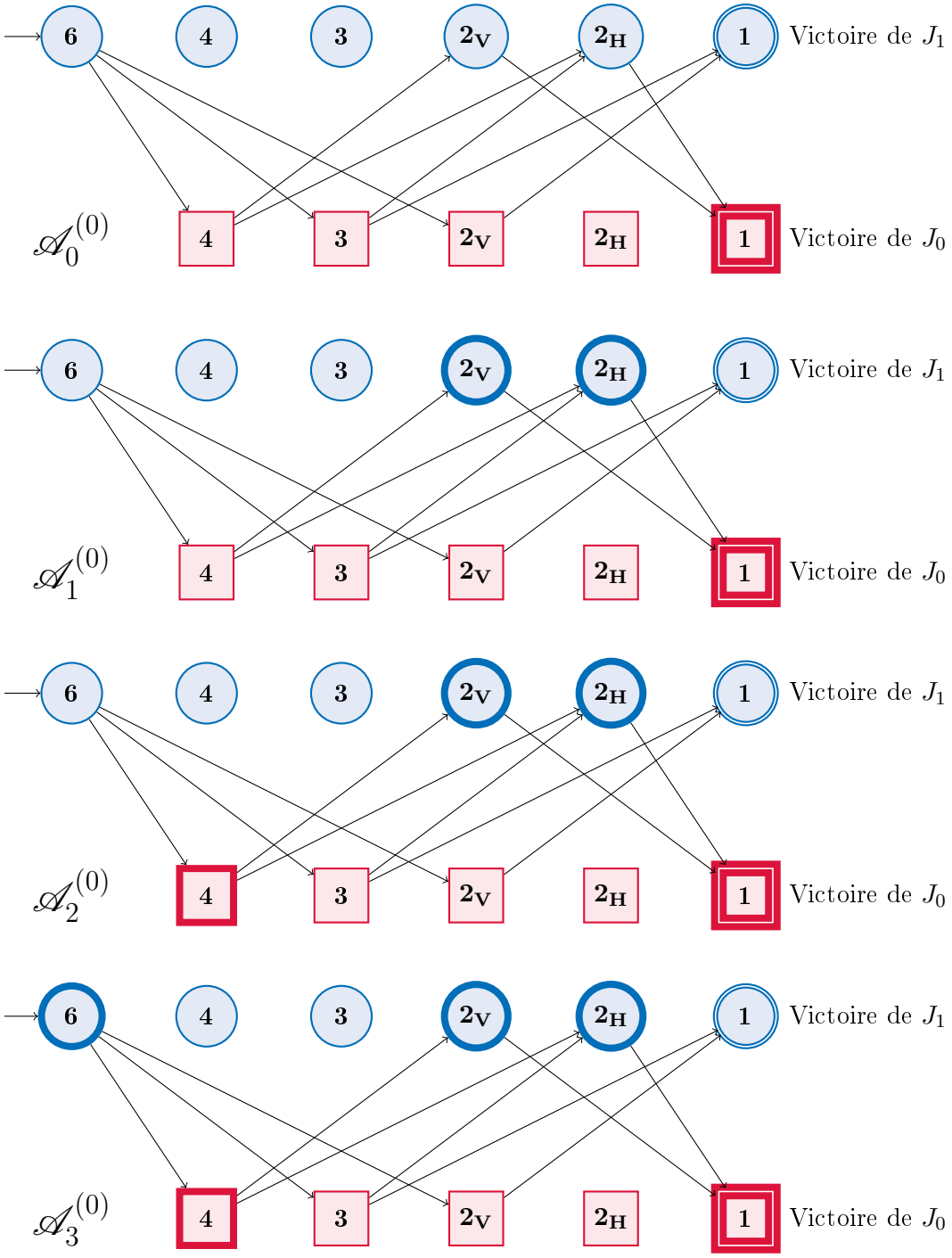


On obtient alors le graphe biparti suivant. On représente les états contrôlés par J_0 par des ronds bleus et les états contrôlés par J_1 par des carrés rouges.



► Déterminer les positions gagnantes pour le joueur J_0.

L'ensemble des positions gagnates pour le joueur J_0 est l'attracteur pour ce joueur. On le note $\mathscr{A}^{(0)}$.
On applique l'algorithme vu en cours pour le déterminer.



De plus : $\mathscr{A}_4^{(0)} = \mathscr{A}_3^{(0)}$. Ainsi, la suite $\left(\mathscr{A}_i^{(0)}\right)$ est stationnaire à partir du rang 3. On en déduit : $\mathscr{A}^{(0)} = \mathscr{A}_3^{(0)}$.

▶ Le joueur J_0 admet-il une stratégie gagnante ? Si oui, déterminer en une.

- Le sommet initial appartient à l'attracteur $\mathscr{A}^{(0)}$. Il existe donc une stratégie gagnante pour le joueur J_0.
- D'après le graphe biparti précédent, une stratégie gagnante est :
  - *(i)* de commencer par ôter la dernière colonne de la tablette. On est alors dans la configuration **4**.
  - *(ii)* après le tour du joueur J_1, on est en situation $\mathbf{2_V}$ ou $\mathbf{2_H}$. Il suffit alors de se ramener à la configuration **1** pour emporter la victoire.

I.2. Étude du cas général

Pour représenter les graphes en **Python**, on utilise un dictionnaire dont les clés sont les sommets du graphe et les valeurs associées sont les listes des successeurs. On représente chaque sommet du graphe biparti $G = (S, A)$ du jeu par un triplet $(a, b, k) \in [\![1, p]\!] \times [\![1, q]\!] \times \{0, 1\}$ où le couple (a, b) est la taille de la tablette restante et k est l'indice du joueur contrôlant cet état du jeu. La fonction ci-dessous permet de générer le graphe biparti du jeu.

```
1  def genGrapheBiparti(p, q) :
2      ''genGraphe(p : int, q : int) -> dico : dict''
3      dico = {}
4      for a in range(1, p+1) :
5          for b in range(1, q+1) :
6              for k in range(2) :
7                  dico[(a,b,k)] = []
8                  for i in range(1, a) :
9                      dico[(a,b,k)].append((i,b,1-k))
10                 for j in range(1, b) :
11                     dico[(a,b,k)].append((a,j,1-k))
12     return dico
```

Par exemple, pour représenter le graphe de la partie précédente, on peut utiliser la commande suivante.

```
1  G = genGrapheBiparti(2, 3)
```

Le graphe transposé G^T d'un graphe $G = (S, A)$ est obtenu en conservant tous les sommets de S et en inversant le sens des arêtes de A. Autrement dit, à la place d'associer à un sommet la liste de ses successeurs, on lui associe la liste de ses prédécesseurs.

▶ Écrire une fonction `transpose` prenant en argument un graphe `G` et renvoie le graphe transposé de `G`.

Dans la suite, on considérera le graphe transposé H du graphe biparti $G = (S, A)$ associé au jeu.

```
1  H = transpose(G)
```

```python
def transpose(G) :
    ''transpose(G : dict) -> dico : dict''
    dico = {sommet : [] for sommet in G}
    for cle1 in dico :
        for cle2 in G :
            if cle1 in G[cle2] :
                dico[cle1].append(cle2)
    return dico
```

Dans ce TP, pour représenter un ensemble E, on utilise un dictionnaire dont les clés sont les éléments de l'ensemble E et les valeurs sont vides. Par exemple, l'ensemble F_0 (respectivement F_1) des sommets finaux correspondant à une victoire de J_0 (respectivement J_1) pour ce jeu se représente de la manière suivante.

```python
F0 = {(1,1,1) : None}
F1 = {(1,1,0) : None}
```

- Construire l'ensemble S_0 (respectivement S_1) des sommets représentant les états du jeu contrôlés par le joueur J_0 (respectivement J_1).

```python
S0 = {(a,b,0) : None for a in range(1, p+1) for b in range(1, q+1)}
S1 = {(a,b,1) : None for a in range(1, p+1) for b in range(1, q+1)}
```

Dans les questions suivantes, nous allons rédiger des fonctions en Python permettant de calculer l'attracteur de chaque joueur.

- Écrire une fonction `fonctAtt` prenant en argument un graphe `G`, une partie `E` de S et un entier `k` $\in \{0, 1\}$ et qui renvoie la partie E' de S définie par :

$$E' = E \cup \left\{ s \in S_k \mid \exists s' \in E,\ (s, s') \in A \right\} \cup \left\{ s \in S_{1-k} \mid \forall s' \in S_k,\ (s, s') \in A \Rightarrow s' \in E \right\}$$

```python
def fonctAtt(G, E, k) :
    ''fonctAtt(G : dict, E : dict, k : int) -> Ep : dict''
    Ep = {sommet : None for sommet in E}
    GT = transpose(G)
    S0 = {sommet : None for sommet in G if sommet[2] == 0}
    S1 = {sommet : None for sommet in G if sommet[2] == 1}
    S = (S0, S1)
    for cle in S[k] :
        for elt in E :
            if cle in GT[elt] :
                Ep[cle] = None
    for cle in S[1-k] :
        TousDansE = True
        for successeur in G[cle] :
            if (successeur in S[k]) and (successeur not in E) :
                TousDansE = False
        if TousDansE :
            Ep[cle] = None
    return Ep
```

- Écrire une fonction `attracteur` prenant en argument un graphe `G` et un entier `k` $\in \{0,1\}$ et qui renvoie l'attracteur du joueur J_k.

```python
def attracteur(G, k) :
    ''attracteur(G : dict, k : int) -> Att : dict''
    Att = {(1,1,1-k) : None}
    while True :
        newAtt = fonctAtt(G, Att, k)
        if newAtt == Att :
            return Att
        Att = newAtt
```

- En utilisant votre fonction précédente, conjecturer l'ensemble des positions gagnantes dans le cas général pour les deux joueurs, puis proposer une stratégie gagnante pour le joueur en admettant une.

- En exécutant la commande `attracteur(A, 0)`, on obtient que les positions gagnantes du joueur J_0 sont :

$$(1,1,1),\ (1,2,0),\ (1,3,0),\ (2,1,0),\ (2,2,1) \quad \text{et} \quad (2,3,0)$$

- En exécutant la commande `attracteur(A, 1)`, on obtient que les positions gagnantes du joueur J_1 sont :

$$(1,1,0),\ (1,2,1),\ (1,3,1),\ (2,1,1),\ (2,2,0) \quad \text{et} \quad (2,3,1)$$

- Comme la configuration initiale $(2,3,0)$ n'appartient qu'à l'attracteur du joueur J_0, on en déduit que seul J_0 admet une stratégie gagnante.
  On peut proposer la stratégie gagnante citée dans une question précédente.

II. Algorithme min-max

Dans cette partie, on s'intéresse au jeu Puissance 4. Le but du jeu est d'aligner une suite de 4 pions de même couleur dans une grille rectangulaire composée de 6 lignes et 7 colonnes. Chaque joueur dispose de 21 pions d'une même couleur. Les deux joueurs placent tour à tour un pion dans la colonne de leur choix, le pion coulisse alors jusqu'à la position la plus basse possible dans ladite colonne, à la suite de quoi c'est à l'adversaire de jouer. Le vainqueur est le joueur qui réalise en premier un alignement (horizontal, vertical ou diagonal) consécutif d'au moins 4 pions de sa couleur. Si toutes les cases de la grille de jeu sont remplies et qu'aucun des deux joueurs n'a réalisé un tel alignement, la partie est déclarée nulle.

Dans la suite, on suppose que J_0 joue avec des pions jaunes et que J_1 joue avec des pions rouges.

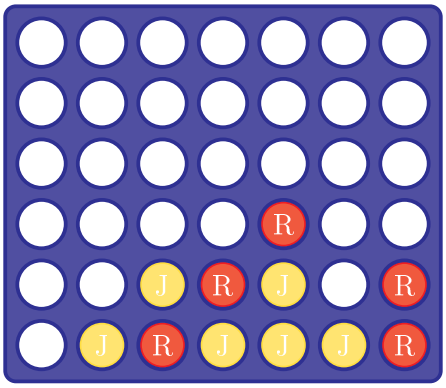


Fig. 1 Une position du jeu Puissance 4

- Pour jouer une partie contre l'ordinateur, vous pouvez utiliser les instructions suivantes.

```
from puissance4 import *
partie()
```

- Gagner une partie contre l'ordinateur.

Normalement, c'était très facile : l'ordinateur joue aléatoirement dans une des colonnes possibles. Nous allons utiliser l'algorithme min-max pour améliorer la stratégie suivie par l'ordinateur.

On commence par construire une heuristique pour ce jeu. Pour ce faire, on attribue à chaque case du jeu le nombre d'alignements possibles de quatre jetons contenant cette case. Les valeurs attribuées sont reportées dans le tableau ci-dessous.

3	4	5	7	5	4	3
4	6	8	10	8	6	4
5	8	11	13	11	8	5
5	8	11	13	11	8	5
4	6	8	10	8	6	4
3	4	5	7	5	4	3

Fig. 2 Tableau des valeurs attribuées à chaque case

Ensuite, pour calculer l'heuristique d'une position (non finale) du jeu, on calcule somme des valeurs des cases contrôlées par J_0 à laquelle on retranche la somme des valeurs des cases contrôlées par J_1.

- Calculer l'heuristique de la position s représentée au début de cette partie.

$$h(s) = (4+7+5+4+8+8) - (5+3+10+4+11) = 3$$

- Quel premier choix doit effectuer le joueur J_0 afin de maximiser l'heuristique après 1 coup ? après 2 coups ?

- Après 1 coup, les heuristiques des positions du jeu sont (de la colonne de gauche à la colonne de droite) :
  $$(6,\ 9,\ 14,\ 16,\ 14,\ 9,\ 8)$$
  Le joueur J_0 devrait donc choisir de jouer dans la $4^{\text{ème}}$ colonne.
- Après 2 coups, les heuristiques des positions du jeu sont (de la colonne de gauche à la colonne de droite) :
  - $(2,\ 0,\ -5,\ -7,\ -6,\ 0,\ 1)$,
  - $(6,\ 1,\ -2,\ -4,\ -2,\ 3,\ 4)$,
  - $(11,\ 8,\ 3,\ 1,\ 3,\ 8,\ 9)$,
  - $(13,\ 10,\ 5,\ 3,\ 5,\ 10,\ 11)$,
  - $(11,\ 8,\ 3,\ 1,\ 6,\ 10,\ 11)$,
  - $(6,\ 3,\ -2,\ -4,\ -2,\ 1,\ 4)$,
  - $(3,\ 0,\ -5,\ -7,\ -5,\ 0,\ 1)$.

  Les heuristiques après le coup du joueur J_1 (qui cherche à minimiser l'heuristique) sont donc :
  $$(-7,\ -4,\ 1,\ 3,\ 1,\ -4,\ -7)$$
  Le joueur J_0 devrait donc choisir là encore de jouer dans la $4^{\text{ème}}$ colonne.

Pour représenter une position du jeu Puissance 4 en Python, on considère une liste de listes dont les éléments sont `'J'` pour case jaune, `'R'` pour case rouge ou `'O'` pour case ouverte. Par exemple, la position dessinée au début de cette partie est représentée par la liste ci-dessous.

```
grille = [
    ['O', 'O', 'O', 'O', 'O', 'O', 'O'],
    ['O', 'O', 'O', 'O', 'O', 'O', 'O'],
    ['O', 'O', 'O', 'O', 'O', 'O', 'O'],
    ['O', 'O', 'O', 'O', 'R', 'O', 'O'],
    ['O', 'O', 'J', 'R', 'J', 'O', 'R'],
    ['O', 'J', 'R', 'J', 'J', 'J', 'R']]
```

Dans le fichier .py du TP, l'heuristique décrite ci-dessus est déjà définie sous le nom `h(grille)`. De même, la fonction `minMax` (prenant en argument une grille, le numéro du joueur et la profondeur d'exploration souhaitée) permettant de mettre en place l'algorithme min-max est déjà rédigée.

- Dans votre fichier **Python** du TP, étudier le fonctionnement puis commenter les lignes du code de la fonction `minMax`.

Pour terminer, on considère une constante indiquant la profondeur d'exploration de l'arbre que devra effectuer le joueur J_1 avec l'algorithme min-max pour choisir son prochain coup.

```
PROFONDEUR_EXPLORATION = 4
```

- Écrire une fonction `IA_minMax` prenant en entrée l'état du jeu représenté par `grille` et la profondeur `profondeur` et qui renvoie l'indice de la colonne que devra jouer l'ordinateur (*i.e.* le joueur J_1) en suivant l'algorithme min-max avec la profondeur `profondeur`.

```
def IA_minMax(grille, profondeur) :
    ''IA_minMax(grille : list(list), profondeur : int) -> int''
    res = minMax(grille, 1, profondeur)
    return res[1]
```

Vous pouvez jouer contre l'ordinateur avec cette nouvelle I.A. en utilisant la commande suivante.

```
partie(IA_minMax)
```

Remarque
On peut augmenter la valeur de `PROFONDEUR` pour améliorer le niveau de l'ordinateur.