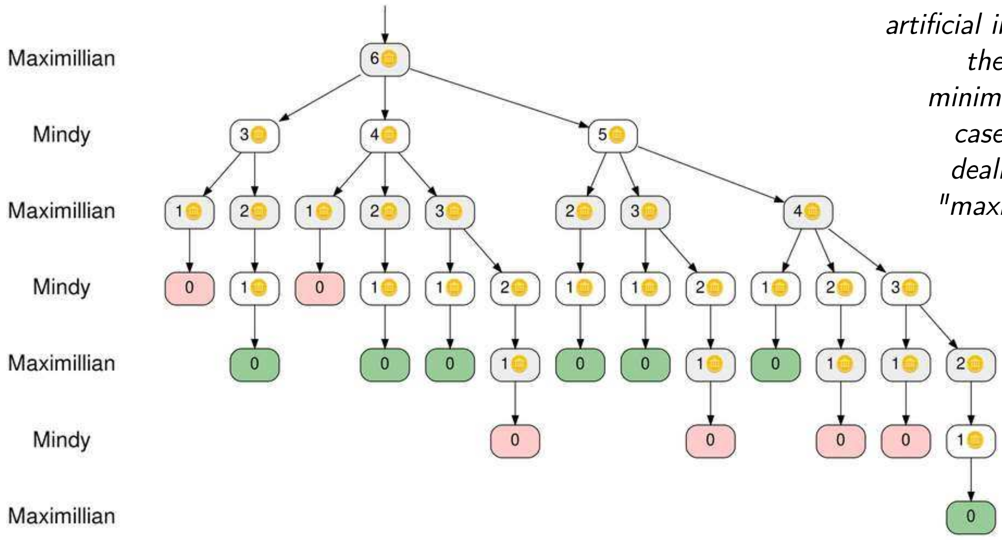
VIII

Algorithme minmax - Utilisation d'une heuristique

« Minimax (sometimes MinMax, MM or saddle point) is a decision rule used in artificial intelligence, decision theory, game theory, statistics, and philosophy for minimizing the possible loss for a worst case (maximum loss) scenario. When dealing with gains, it is referred to as "maximin"- to maximize the minimum gain. » Source : en.wikipedia.org



PLAN DU CHAPITRE

I Algorithme du minmax

I.1 Représentation d'un jeu par un arbre

Le chapitre précédent proposait d'étudier les jeux en les représentant par des graphes et en parcourant ces derniers pour dégager des stratégies gagnantes.
On se propose ici de représenter les différents états du jeux par un **arbre** également dans le but de **dégager des stratégies gagnantes**.

Définition I-1: ARBRE

Un arbre est un graphe connexe, sans cycle, orienté de haut en bas, comportant :
- un sommet unique appelé **racine** ne possédant **aucun prédécesseur** ;
- tous ses autres sommets admettant pour chacun **un prédécesseur unique**.

Un sommet ne possédant pas de successeurs est nommé **feuille** ou **nœud terminal**.

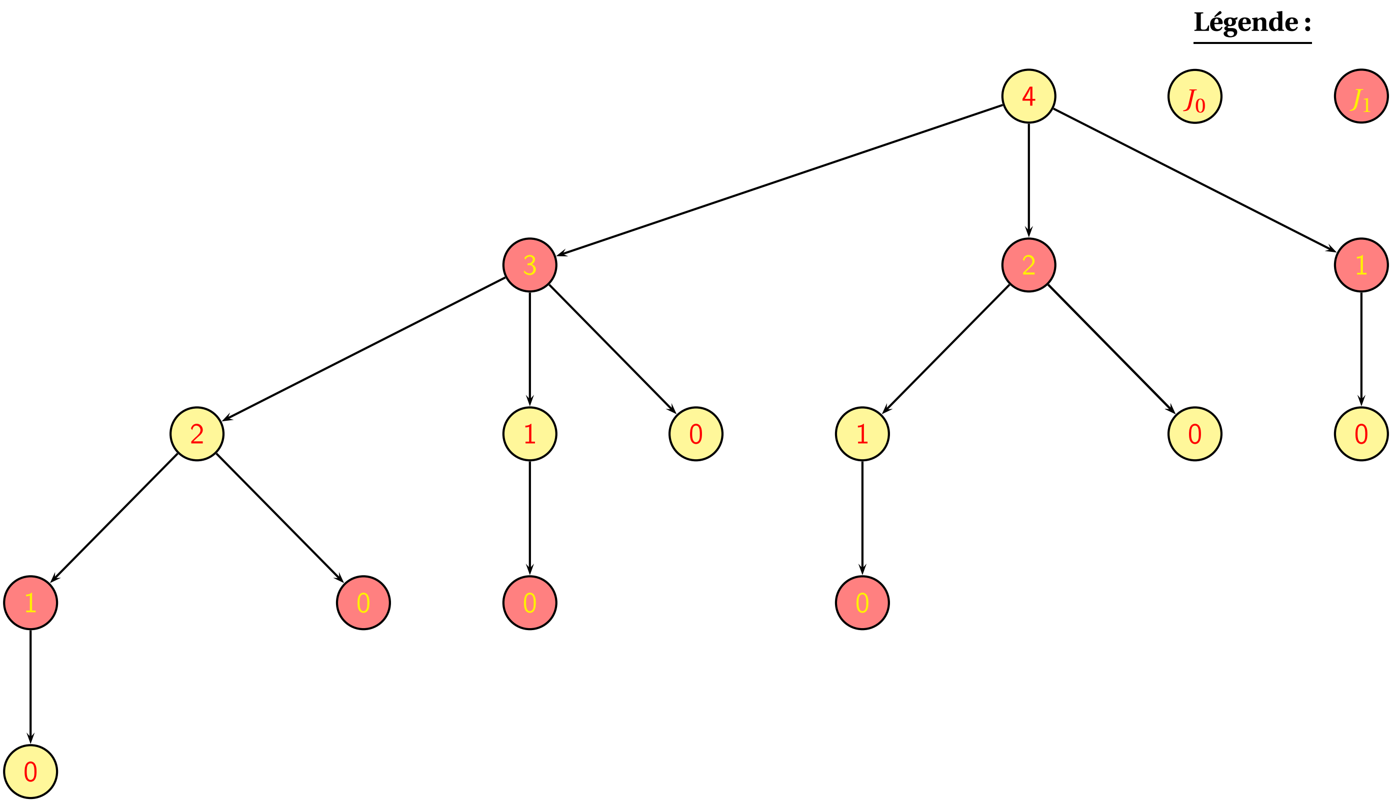
Définition I-2: PROFONDEUR D'UN SOMMET

La profondeur d'un sommet dans un arbre est sa distance verticale à la racine de l'arbre (donc le nombre de ses ascendants) ; la profondeur de la racine est donc 0.

Propriété I-1: UNICITÉ D'UN CHEMIN VERS LA RACINE

Le fait qu'un arbre soit connexe et sans cycle entraine qu'il existe un unique chemin permettant de remonter d'un noeud quelconque à la racine

Par exemple, l'arbre représentant une partie du jeu de Nim avec $N = 4$ jetons pour lequel chaque joueur peut retirer 1,2, ou 3 jetons est :



Remarque I-1: IMPORTANT !

- Dans cet exemple, les sommets sur fond jaune sont contrôlés par J_0 et ceux sur fond rouge par J_1 ; c'est donc ici le joueur J_0 **qui démarre la partie**.

- On constate que des **sommets distincts au même niveau dans l'arborescence** peuvent représenter **la même situation du jeu**, i.e. **le même nombre de jetons restants, et le contrôle par le même joueur**. Il faudra nécessairement les distinguer dans l'implémentation de l'arbre (cf plus bas)

I.2 Algorithme du minmax : étiquetage de l'arbre

Avec la représentation en arbre du jeu, on constate que les sommets terminaux (feuilles) représentent nécessairement le résultat d'une partie, à savoir : partie gagnée par J_0, ou partie gagnée par J_1 (, ou match nul, ce dernier état n'existant pas dans le jeu de Nim).

PRINCIPE DE L'ALGORITHME DU MINMAX :

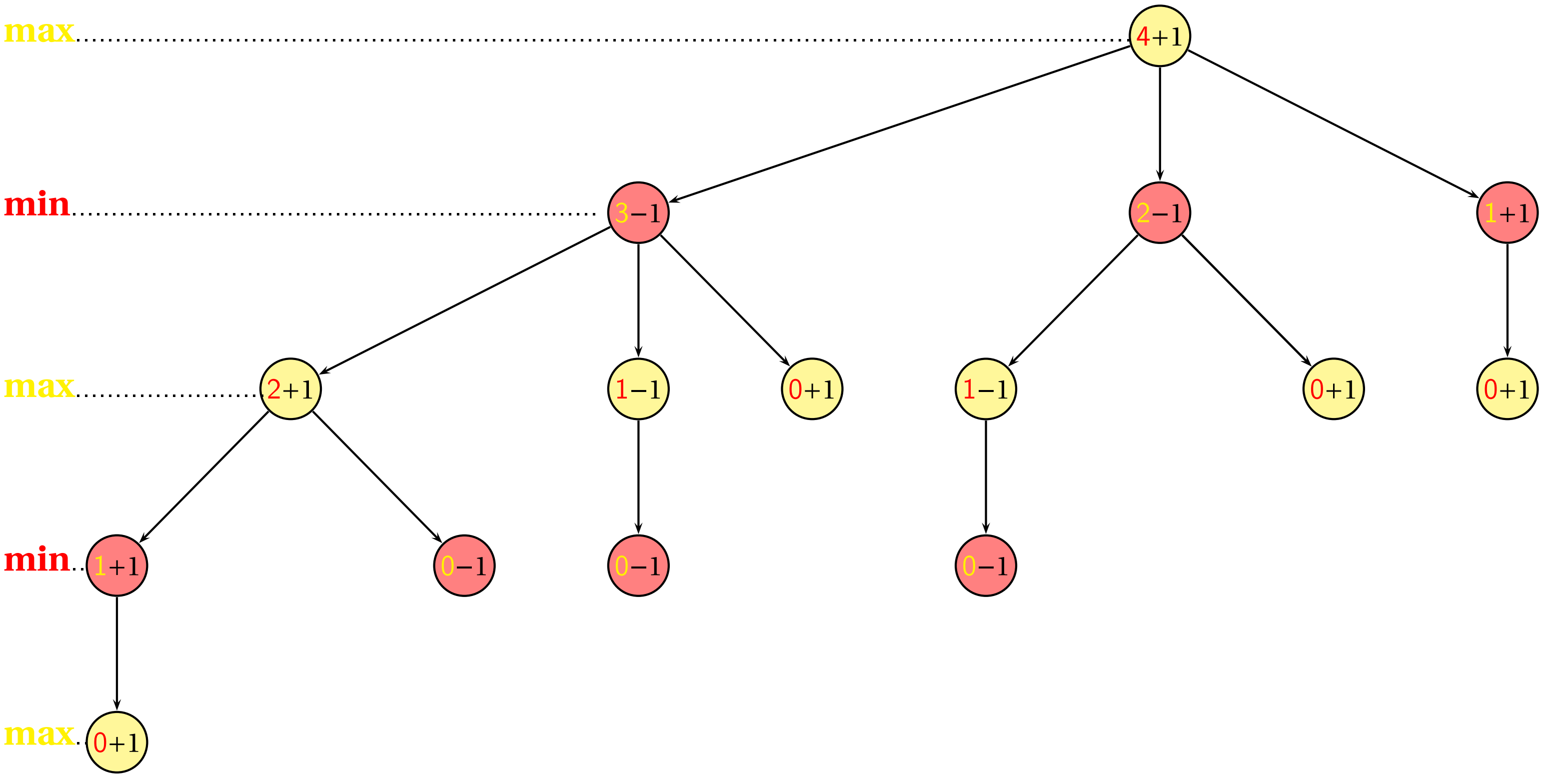
Dans l'**algorithme du minmax**, on attribue à ces sommets terminaux (feuilles) une **étiquette** qui sera :

- $+1$ si J_0 est le gagnant
- -1 si J_1 est le gagnant
- 0 si le match est nul (ce qui n'arrive pas dans le jeu de Nim)

puis on va définir de manière ascendante les **étiquettes** des autres sommets de l'arbre selon le principe suivant :

- si le sommet est contrôlé par J_0 alors son étiquette sera le **maximum** des valeurs de ses sommets fils
- si le sommet est contrôlé par J_1 alors son étiquette sera le **minimum** des valeurs de ses sommets fils

Cela donne le graphe suivant pour la partie du jeu de Nim à 4 jetons :



> <u>Observations :</u>
>
> Si J_0 (respectivement J_1) se trouve sur un sommet **max** (respectivement **min**) d'étiquette $+1$ (respectivement -1), alors :
>
> - il devra choisir le ou l'un des fils de valeur $+1$ (respectivement -1) pour son prochain coup ;
> - alors : le joueur J_1 (respectivement J_0) qui a le contrôle de ce sommet **min** n'aura pas d'autre choix que de choisir l'un des deux sommets d'étiquette $+1$ (respectivement -1).
> - cette même situation se reproduit jusqu'au sommet terminal qui sera un sommet **max** de valeur $+1$: J_0 **gagne la partie** (respectivement un sommet **min** de valeur -1 : J_1 **gagne la partie**).

I.3 Implémentation de l'arbre

On va maintenant implémenter l'arbre par un dictionnaire ; les clés désigneront les sommets et seront des chaines de caractères comportant : le type de sommet, à savoir **max** s'il est contrôlé par J_0, ou bien **min** si c'est par J_1, le nombre de jetons restants $n^0 sommet$, et enfin $idtsommet$ un entier identifiant de façon unique le sommet (ceci afin de distinguer les sommets décrivant des situations identiques de jeux à la même profondeur de l'arbre) :

$$\text{clé : } \boxed{'type\_n^0 sommet - idtsommet'}$$

Par exemple pour le sommet racine de l'arbre ci-dessus contrôlé par J_0 et à 4 jetons : **'max_4-0'**

La valeur attachée à chaque clé sera la liste des sommets fils (sous forme de clés comme décrit ci-dessus) :

valeur : $\boxed{'[clé\_fils_1,\ clé\_fils_2,\ .....]''}$

Afin de pouvoir par la suite vérifier le bon fonctionnement de nos algorithmes, on va implémenter l'arbre correspondant au jeu de Nim. On propose pour cela le code suivant :

Listing VIII.1 –

```
def constrarbreNIM(N,arbre,j,idt):
    #N: nombre de jetons au départ
    #arbre: le dictionnaire des sommets
    #j le joueur 0 ou 1
    #idt l'identificateur de sommet
    def labelize(nature,n,idt): #fabrication des clés
            return "{0}_{1}-{2}".format(nature,n,str(idt))
    nature=["max","min"]
    sommet=labelize(nature[j],N,idt) #crée la clé du sommet selon le schéma vu plus haut
    arbre[sommet]=[] #initialise une liste vide comme valeur du noeud dans le dictionnaire

    for i in [1,2,3]:
        if N-i>=0:
            fils=labelize(nature[1-j],N-i,len(arbre)+i)
            arbre[sommet].append(fils)
            constrarbreNIM(N-i,arbre,1-j,len(arbre)+i)
    return arbre
```

A titre d'exemple, on donne ici la sortie renvoyée par ce code pour $N = 4$ jetons ; le lecteur pourra vérifier sa conformité avec le graphe présenté plus haut.

```
>>> arbre=constrarbreNIM(4,{},0,0)
>>> print("Dictionnaire de l'arbre :",arbre)

Dictionnaire de l'arbre : 'max_4-0' : ['min_3-2', 'min_2-11', 'min_1-16'], 'min_3-2' : ['max_2-3', 'max_1-8', 'max_0-11'], 'max_2-3' : ['min_1-4', 'min_0-7'], 'min_1-4' : ['max_0-5'], 'max_0-5' : [], 'min_0-7' : [], 'max_1-8' : ['min_0-8'], 'min_0-8' : [], 'max_0-11' : [], 'min_2-11' : ['max_1-11', 'max_0-14'], 'max_1-11' : ['min_0-12'], 'min_0-12' : [], 'max_0-14' : [], 'min_1-16' : ['max_0-15'], 'max_0-15' : []
```

I.4 Implémentation de l'algorithme

On implémente enfin l'algorithme qui va parcourir récursivement l'arbre en choisissant le mininum ou le maximum pour chaque sommet fils suivant le joueur qui contrôle le sommet père.
Lorsque l'on arrive sur un sommet terminal (sans fils, donc plus de jetons à retirer) :

- si c'est un sommet **max** (i.e. contrôlé par J_0) alors on renvoie la valeur $+1$ ce qui signifie que J_0 remporte la partie ;
- et si c'est un sommet **min** (i.e. contrôlé par J_1) alors on renvoie la valeur -1 ce qui signifie que J_1 remporte la partie.

Listing VIII.2 –

```
def minmax(sommet,arbre):
```

```
    if arbre[sommet]==[]:
        if ("max" in sommet): #si ce sommet est contrôlé par J0 il gagne la partie
            return +1 #
        else: #sinon il est contrôlé par J1 qui gagne la partie
            return -1 #
    else: #sinon on lance la récursion minmax
        if ("max" in sommet): #si c'est un sommet contrôlé par J0
            return max([minmax(fils, arbre) for fils in arbre[sommet]]) # fils d'étiquette maxi.
        else: # sinon c'est un sommet contrôlé par J1
            return min([minmax(fils,arbre) for fils in arbre[sommet]]) # fils d'étiquette mini.
```

Par exemple, si J_0 démarre une partie avec 4 jetons, l'appel initial sera : `minmax('max_4-0',arbre)` ; l'exécution du code donne :

```
>>> print(minmax('max_4-0',arbre))
1
```

ce qui correspond bien à l'étiquette de la racine de l'arbre représenté plus haut.

INTERPRÉTATION :

- Lorsqu'un sommet contrôlé par J_0 (respectivement J_1) est à la valeur $+1$ (respectivement -1), cela signifie que J_0 (respectivement J_1) peut imposer la suite des coups conduisant à sa victoire.
- En particulier, si le sommet racine est contrôlé par J_0 et qu'il est à la valeur est $+1$, J_0 peut trouver une stratégie assurant sa victoire sur cette partie.

Comme ici le premier joueur était J_0 et que la valeur de sortie est $+1$ pour le sommet racine, alors il va remporter la partie s'il suit la stratégie du "minmax".

II Algorithme minmax avec fonction heuristique

II.1 Principe

Dans le jeu de Nim, l'arbre comporte un nombre de sommets qui progresse de manière exponentielle avec le nombre de jetons de départ ; pour certains jeux, comme les échecs pour lesquels les possibilités de coups sont très nombreuses, la construction de l'arbre du jeu, alors immense, ainsi que son parcours d'étiquetage ne sont clairement pas envisageables ; l'algorithme `minmax` n'est donc plus exploitable.

L'idée va ici consister à n'explorer qu'une partie de l'arbre en s'intéressant à des sommets (états du jeu) qui semblent "plus favorables", et on ne cherchera pas non plus à explorer l'arbre sur toute sa profondeur. On parlera là de **méthodes heuristiques** ou simplement d'**heuristiques**.

Définition II-1: HEURISTIQUE

Une heuristique est une méthode approchée, non nécessairement optimale, mais qui fournit plus rapidement qu'un algorithme rigoureux prouvé (alors inefficace) des **solutions acceptables**.

PRINCIPE DE L'ALGORITHME :

- On va supposer connue une fonction d'évaluation "heuristique" $h(sommet)$ qui renvoie un **score** pour le sommet `sommet` : plus celui-ci est élevé, plus la situation du joueur J_0 est favorable. Par exemple, pour une partie d'échecs, cette valeur sera fonction des différentes pièces et de leurs positions sur l'échiquier.

- On va partir d'une certaine position dans l'arbre à la profondeur p_i, et explorer une profondeur $prof$ à partir de celle-ci, c'est à dire examiner $p_i \ldots ...p_i + prof$

  L'algorithme **minmax avec heuristique** est très proche de **minmax vu plus haut** ; on parcourt l'arbre récursivement entre p_i et $p_i + prof$ et :
- si l'on tombe sur un sommet terminal avant d'avoir atteint la profondeur maximale, alors on attribue +1 ou -1 suivant la nature du sommet **max/min** et on déclare le joueur concerné vainqueur ;
- si l'on a atteint la profondeur maximale, on renvoie la valeur donnée par la fonction heuristique sur ce sommet

Le code modifié est donc :

Listing VIII.3 –

```
def minmax(sommet, arbre, prof, h):
  if arbre[sommet]==[]:
    if ("max" in sommet): #si ce sommet est contrôlé par J0 il gagne la partie
      return +1 #
    else: #sinon il est contrôlé par J1 qui gagne la partie
      return -1 #
  elif prof==0:
    return h(sommet)
  elif ("max" in sommet): #si c'est un sommet contrôlé par J0
      return max([minmax(fils, arbre,prof-1,h) for fils in arbre[sommet]]) # fils d'étiqu. maximale
  else: # sinon c'est un sommet contrôlé par J1
      return min([minmax(fils,arbre,prof-1,h) for fils in arbre[sommet]]) # fils d'étiqu. minimale
```

II.2 Exemple : heuristique dans le jeu puissance 4

Dans le jeu *Puissance 4*, les deux joueurs jouent à tour de rôle en laissant tomber dans l'une des colonnes d'une grille (6×7) chacun de leur pion (identifié par une couleur), qui vient alors se positionner sur la case libre la plus basse de cette colonne ; le premier joueur **qui aligne 4 pions remporte la partie**.

Une partie en cours avec les pions jaunes pour le joueur $J0$ et rouges pour $J1$ est par exemple :

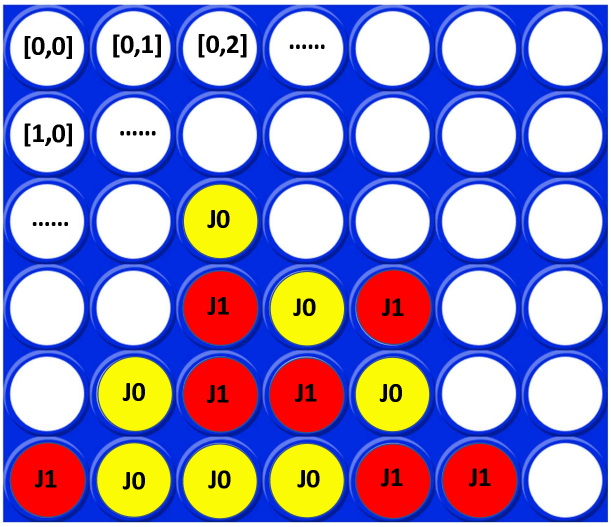


FIGURE VIII.1 – Une partie en cours !

Naturellement, les cases situées vers le centre de la grille appartiennent à un nombre plus importants d'alignements potentiels de 4 pions. Il est donc plus judicieux de jouer "au centre". Par conséquent une **heuristique simple** pour ce jeu consiste à attribuer à chaque case le nombre d'alignements de 4 pions auquel elle peut participer ; chaque joueur cherchera alors à placer ses pions sur des cases à forte « cotation ».

La valeur de l'heuristique sera **la somme des cotations des cases occupées par les pions de J_0 moins la somme des cotations de celles occupées par les pions de J_1.**

CODES PYTHON :

- On commence par coder la fonction permettant de calculer les cotations de chaque case de la grille :

Listing VIII.4 – fonction de cotation de la grille

```
def cotation_grille():
    n,m=6,7 #grille de 6 lignes et 7 colonnes
    grille_cotee=np.zeros((n,m), dtype=int) #tableau de zeros
    for i in range(n): #itérations sur les lignes
        for j in range(m): #itérations sur les colonnes
            for (x,y) in [(1,0),(0,1),(1,1),(1,-1)]: #choix de la direction de parcours
                if (0<=i+(4-1)*x<n) and (0<=j+(4-1)*y<m): #vérif. appart. case à la grille
                    for k in range(4): #itérations sur le nombre de jetons alignés
                        grille_cotee[i+k*x,j+k*y]=grille_cotee[i+k*x,j+k*y]+1 #incrém. cote
    return grille_cotee
```

ce qui donne :

```
>>> print(cotation_grille())
[[ 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]]
```

soit plus "graphiquement" :

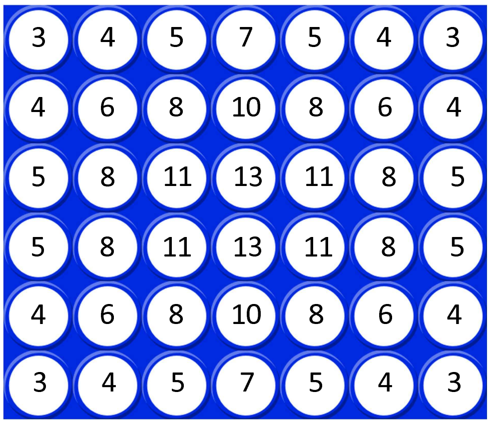


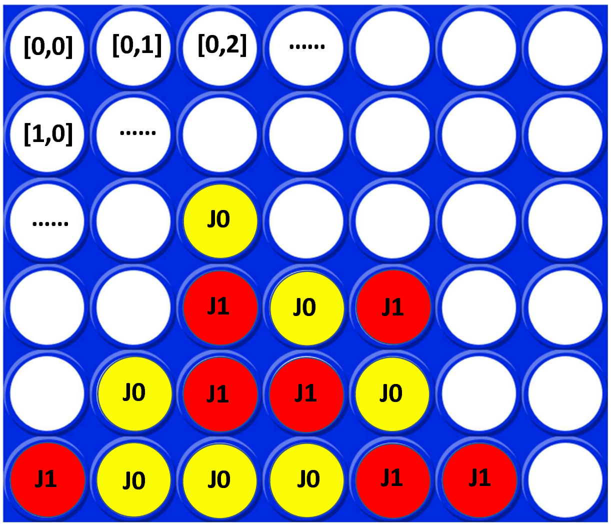
FIGURE VIII.2 – Grille puissance 4 « cotée »

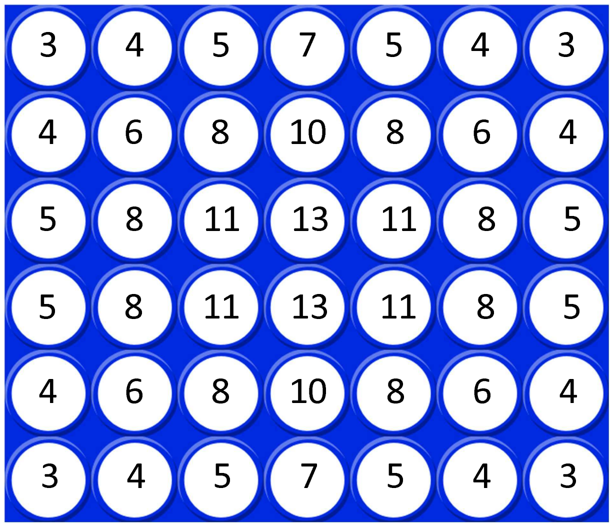
- puis la fonction calculant la valeur l'heuristique pour une situation de jeu donnée :

Listing VIII.5 – Calcul de la valeur de l'heuristique

```python
def h(etat_grille:array):
    grille_cotee=cotation_grille()
    n,m=etat_grille.shape
    S=0
    for i in range(n):
        for j in range(m):
            if "J0" in etat_grille[i,j]:
                S=S+grille_cotee[i,j]
            if "J1" in etat_grille[i,j]:
                    S=S-grille_cotee[i,j]
    return S
```

En reprenant par exemple l'état du jeu présenté en figure 1 que l'on rappelle ici (ainsi que la grille cotée) :





la valeur de l'heuristique est alors :

$$h(grille) = [4+5+7+6+8+13+11] - [3+5+4+8+10+11+11] = 2$$