Algorithme minmax

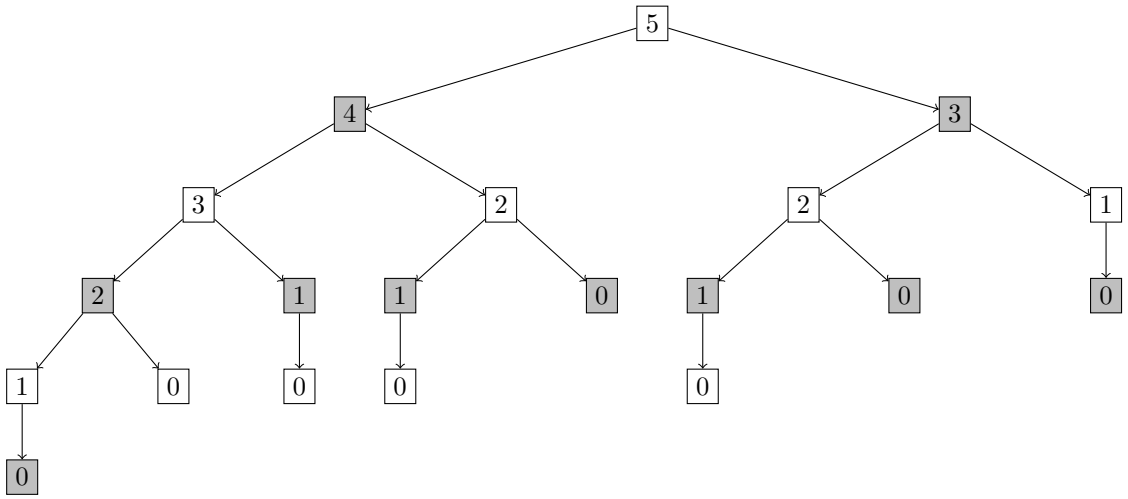
I Représentation par un arbre

1. Arbre du jeu

Pour introduire l'algorithme minmax, dont le but est de déterminer une stratégie gagnante, on va représenter les situations du jeu par un arbre :
Un arbre est un graphe connexe (en une seule partie) et qui ne contient pas de cycle. On distingue deux types parmi les sommets du graphe :

- les feuilles qui sont des sommets n'ayant pas de descendants
- les nœuds qui désignent les autres sommets.

On appelle racine de l'arbre un des nœud qui va en fait servir à repérer les autres : la profondeur d'un nœud est le nombre d'arêtes que l'on doit utiliser pour le rejoindre depuis la racine. La racine est donc le seul nœud de profondeur 0, ceux de profondeur 1 sont ceux qui sont directement liés à la racine.
Dans un arbre représentant un jeu, la racine correspond à la situation de départ, chaque nœud est une de situations possibles et les feuilles sont les situation pour lesquelles on ne peut plus jouer.
Voici par exemple un arbre représentant le jeu de Nim à un seul tas, comportant 5 jetons au départ et dans lequel on peut retirer 1 ou 2 jetons à chaque fois.



Si le joueur qui joue en premier est J_1, les nœeuds blancs sont les positions contrôlées par J_1, les nœuds grisées sont les positions contrôlées par J_2. Les positions contrôlées par J_1 sont donc les nœuds de profondeur paire, celles contrôlées par J_2 sont de profondeur impaire.
On peut remarquer qu'il est possible d'avoir deux nœuds au même niveau, correspondant à la même situation de jeu, ie avec le même nombre de jetons restant. Pour distinguer deux tels nœuds, on va rajouter un paramètre pour les représenter de façon unique. Dans la suite, on supposera que l'arbre est implémenté par un dictionnaire `arbre`, dont les clefs sont les nœuds, de la forme `(p,n,k)` (avec `p` la profondeur du nœud, `n` le nombre de jetons restant et `k` un paramètre pour distinguer d'éventuels nœuds « égaux » à une telle profondeur), et la valeur associée est la liste des fils.

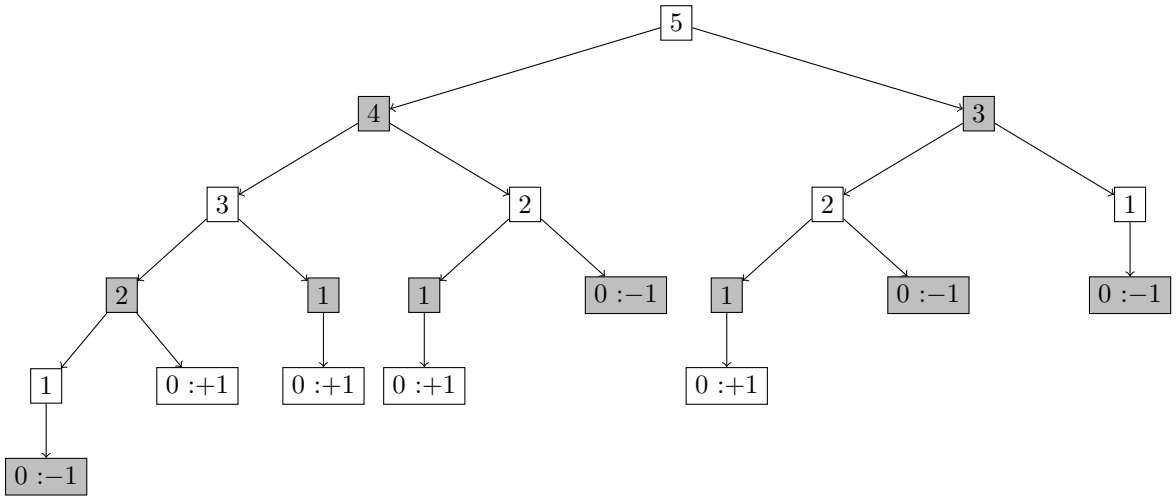
2. Étiquetage de l'arbre

On va attribuer une valeur numérique, appelée étiquette, à chaque nœud de l'arbre. Pour cela, on va considérer que l'on cherche une stratégie gagnante pour J_1 et on va commencer par étiqueter les feuilles :

— si une feuille correspond à une victoire de J_1, on va l'étiqueter $+1$; sur notre exemple, ce sont donc les feuilles à une profondeur paire.

— si une feuille correspond à une victoire de J_2, on l'étiquette -1 ; celles à une profondeur impaire sur notre exemple.

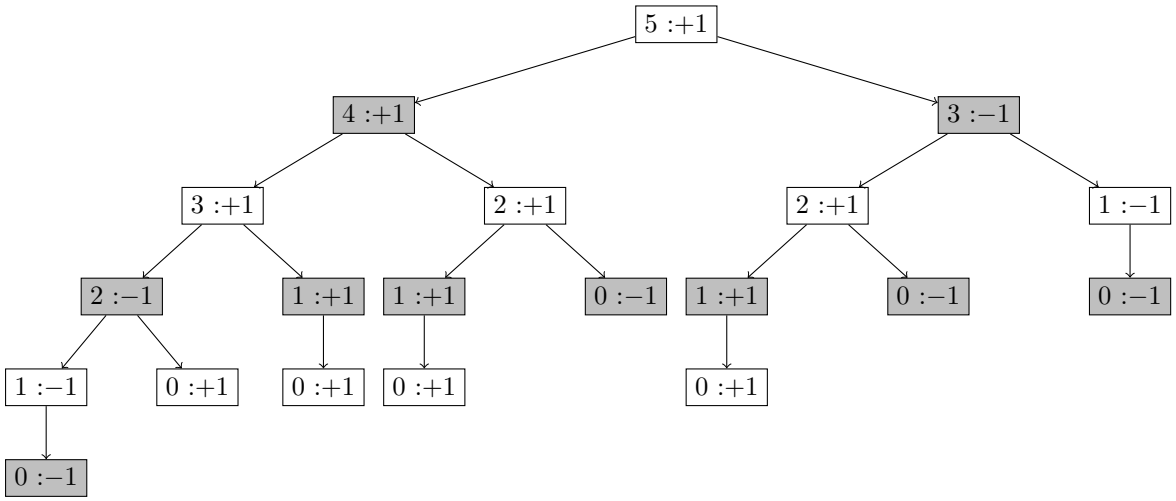
— si une feuille correspond à une situation de match nul (qui n'arrive pas au jeu de Nim), on l'étiquette 0



Étiquetage des feuilles

On définit alors l'étiquette des autres nœuds récursivement :

- pour un nœud contrôlé par J_1, l'étiquette est la valeur maximale des étiquettes de ses fils.
- pour un nœud contrôlé par J_2, l'étiquette est la valeur minimale des étiquettes de ses fils.



Étiquetage des nœuds

On peut alors coder récursivement la fonction qui renvoie pour un nœud de l'arbre son étiquette :

```
def MinMax(noeud,arbre) :
    p,n,k = noeud
    if arbre[noeud] == [] :
        if p%2 == 0 :          # contrôlé par J1, J2 vient de perdre
            return +1
        else :                 # contrôlé par J2, J1 vient de perdre
            return -1
    elif p%2 == 0 :            # contrôlé par J1
        return max([MinMax(fils,arbre) for fils in arbre[noeud]])
    else :                     # contrôlé par J2
        return min([MinMax(fils,arbre) for fils in arbre[noeud]])
```

Pour reconstruire la stratégie gagnante pour J_1, il suffirait alors, en partant de la position initiale, de chercher parmi les fils celui dont l'étiquette est la plus grande. Pour éviter de devoir recalculer l'étiquette de chaque nœud à chaque fois, on pourrait modifier la fonction `MinMax` pour qu'elle remplisse par exemple un dictionnaire dont les clefs sont les nœuds et la valeur associée son étiquette.

II Utilisation d'une heuristique

1. Recherche à une profondeur fixée

Même si les jeux que l'on considère sont des jeux finis (ils se termine toujours, éventuellement par un match nul au bout d'un nombre fini de coups), l'arbre du jeu peut avoir une profondeur très grande et le nombre de nœud est alors exponentiel par rapport à cette profondeur. Le calcul de toutes les étiquettes en partant des feuilles serait alors beaucoup trop long.
On va alors limiter l'exploration de l'arbre à partir d'une position donnée à une certaine profondeur p, ce qui revient à essayer de prévoir ce qui peut se passer au cours des p coups suivants.
Si on note h la profondeur du nœud initial, on devrait descendre jusqu'aux feuilles dans sa descendance pour calculer son étiquette mais pour les atteindre on peut avoir besoin de plus de p coups. On suppose donc avoir une fonction `H`, appelée heuristique, qui est un moyen d'estimer si la position est gagnante pour J_1 ou non : pour un nœud `n` donnée, plus la valeur de `H(n)` est grande, plus la situation est favorable pour J_1.
On peut alors définir l'étiquettes des nœuds `n` aux profondeurs comprises entre h et $p+h$:

- si `n` est une feuille, on lui attribue la valeur $+1000$ ou -1000 selon le joueur vainqueur comme précédemment (la valeur 1000 est choisie en supposant qu'elle est, en valeur absolue toujours supérieure aux valeurs de l'heuristique).
- si `n` est à la profondeur $p+h$ mais n'est pas une feuille, on lui attribue la valeur `H(n)`.
- pour les nœuds aux profondeurs inférieures, on calcule leur étiquette comme précédemment : le maximum des étiquettes de ses fils si `n` est contrôlé par J_1, le minimum s'il est contrôle par J_2.

On modifie alors le code de la façon suivante :

```
def MinMax(noeud,arbre,prof,H) :
    p,n,k = noeud
    if arbre[noeud] == [] :
        if p%2 == 0 :          # contrôlé par J1, J2 vient de perdre
            return +1000
        else :                 # contrôlé par J2, J1 vient de perdre
            return -1000
    elif prof == 0 :           # on a atteint la profondeur de recherche
        return H(noeud)
    elif p%2 == 0 :            # contrôlé par J1
        return max([MinMax(fils,arbre,prof-1,H) for fils in arbre[noeud]])
    else :                     # contrôlé par J2
        return min([MinMax(fils,arbre,prof-1,H) for fils in arbre[noeud]])
```

2. Un exemple d'heuristique

On considère le jeu de « Puissance 4 » (dont le but est d'aligner dans une grille 4 jetons de sa couleur, horizontalement, verticalement ou en diagonale).

		2				
		2				
		1	2	1		
	2	1	1	2		
1	2	2	2	1	1	1

Une situation à Puissance 4

On commence par dénombrer le nombre d'alignement d'au moins 4 jetons de même couleur passant par chaque case. On obtient alors le tableau suivant :

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

On calcule alors l'heuristique d'une position en sommant les valeurs du tableau, avec le signe $+$ pour les cases occupées par un jeton de J_1, et un signe $-$ pour les cases occupées par un jeton de J_2.
Pour la situation dessinée précédemment, la valeur de l'heuristique vaudrait

$$3-4-5-7+5+4+3-6+8+10-8+11-13+11-11-8=-7$$