TD10 : Algorithme MinMax (V3)

- Référence : ce TD est inspiré de la source que l'on trouvera ici
- Code du notebook associé : **267c-1479629**

1 Implémentation du jeu de puissance 4

1.1 Description du jeu

Le jeu se joue à deux joueurs, sur un plateau – posé verticalement, composé de 7 colonnes de hauteur 6.

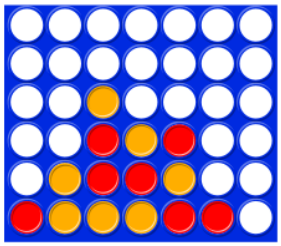


Figure 1

Chaque joueur dispose de jetons (ici jaunes pour le joueur 1, rouges pour le joueur 2).
Les joueurs jouent à tour de rôle, et l'on convient ici que le joueur 1 est le premier à jouer.
Lorsque son tour de jeu est arrivé, un joueur choisit l'une des sept colonnes pour y déposer un de ses jetons : le jeton déposé vient se positionner dans la colonne choisie, au-dessus des jetons déjà présents.
Un joueur gagne lorsque le jeton qu'il vient de déposer réalise un alignement de quatre jetons adjacents de même couleur, soit verticalement, soit horizontalement, soit selon une « diagonale » ascendante ou descendante.

Note : On peut constater que ce jeu rentre bien dans le cadre du programme, qui s'intéresse à des jeux à deux joueurs, dans lesquels chaque joueur joue à tour de rôle, avec une connaissance complète de l'état du jeu (jeux *à information complète*), avec pour but d'atteindre un état du jeu qui permet de le déclarer gagnant (jeux d'*accessibilité*).
Les jeux considérés sont « sans mémoire », dans le sens où l'historique des coups précédents ne sera pas pris en compte, et se déroulent en temps « infini » (chaque joueur dispose du temps qu'il souhaite pour jouer un coup).

1.2 Modélisation du plateau de jeu

On définira deux variables globales définissant le nombre de rangées et de colonnes du plateau :

```
nrows, ncols = 6, 7
```

On modélise le plateau de jeu par un dictionnaire de la forme suivante :

```
jeu = {'nbcoups': ..., 'hauteurs': ..., 'plateau': ...}
```

La clé `'nbcoups'` est associée au nombre de coups joués depuis le début de la partie.
La clé `'hauteurs'` est associée à la liste des nombres de jetons présents dans chaque colonne.
La clé `'plateau'` est associée à un tableau d'entiers à deux dimensions représentant l'état du plateau de jeu.

Les sept colonnes du plateau de jeu sont numérotées de 0 à 6, et les six rangées sont numérotées de 0 à 5, la rangée inférieure étant numérotée 5.
En conséquence, chaque case du plateau est repérée par un couple de coordonnées, (i, j), i et j désignant respectivement le numéro de ligne et de la colonne sur lesquelles se situe la case.

i \ j	0	1	2	3	4	5	6
0	(0, 0)	(0, 1)	(0, 2)	(0, 3)	(0, 4)	(0, 5)	(0, 6)
1	(1, 0)	(1, 1)	(1, 2)	(1, 3)	(1, 4)	(1, 5)	(1, 6)
2	(2, 0)	(2, 1)	(2, 2)	(2, 3)	(2, 4)	(2, 5)	(2, 6)
3	(3, 0)	(3, 1)	(3, 2)	(3, 3)	(3, 4)	(3, 5)	(3, 6)
4	(4, 0)	(4, 1)	(4, 2)	(4, 3)	(4, 4)	(4, 5)	(4, 6)
5	(5, 0)	(5, 1)	(5, 2)	(5, 3)	(5, 4)	(5, 5)	(5, 6)

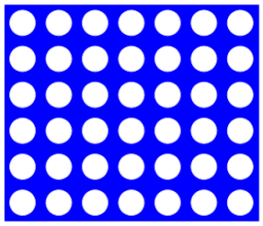
Q1. Définir les variables globales `nrows` et `ncols`.

Q2. Définir une fonction `init()` renvoyant un jeu dans son état initial.
On convient que pour une case inoccupée, on inscrira la valeur zéro dans le tableau `jeu['plateau']` et le numéro du joueur l'occupant sinon.
On devra donc avoir :

```
>>> jeu = init()
>>> jeu
{'nbcoups': 0, 'hauteurs': [0, 0, 0, 0, 0, 0, 0], 'plateau': [[0, 0, 0, 0, 0, 0, 0], [0, 0,
0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0], [0, 0,
0, 0, 0, 0, 0]]}
```

On dispose de deux fonctions, `affiche(jeu)` et `image(jeu)`, affichant, respectivement, l'état du plateau de jeu sous les deux formes suivantes :

```
>>> jeu = init()
>>> affiche(jeu)
0| _ _ _ _ _ _ _
1| _ _ _ _ _ _ _
2|
3| _ _ _ _ _ _ _
4| _ _ _ _ _ _ _
5| _ _ _ _ _ _ _
   0 1 2 3 4 5 6
```



Q3. Définir une fonction `copie(jeu)` renvoyant une copie du jeu. On prêtera attention au fait d'effectuer des copies des listes et des listes de listes, qui soient bien indépendantes des listes et listes de listes originales.

1.3 Modélisation du déroulement du jeu

Q4. Définir une fonction `prochain_joueur(jeu)`, renvoyant le numéro du joueur devant jouer le coup suivant si le jeu est dans l'état décrit par `jeu`.

Q5. Définir une fonction `colonnes_jouables(jeu)`, renvoyant, sous la forme d'une liste de numéros de colonnes, les colonnes dans lesquelles un jeton peut être ajouté si le jeu est dans l'état décrit par `jeu`.

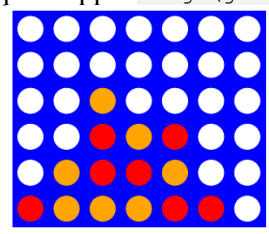
Q6. Définir une fonction `jouer_colonne(jeu, joueur, col)`, qui joue un coup dans la colonne `col` pour le joueur `joueur`. La fonction agira par effet, en modifiant l'état du dictionnaire `jeu`, et renverra les coordonnées de la case jouée : on inscrira, sur le plateau de jeu, le numéro du joueur (1 ou 2) dans la case occupée par le jeton joué, et on actualisera le nombre total de coups joués et les nombres de jetons dans les différentes colonnes. On vérifiera, au titre de préconditions et à l'aide de `assert`, que le joueur `joueur` est bien le prochain joueur à jouer et que la colonne `col` n'est pas pleine.

Q7. Écrire de même une fonction `dejouer_colonne(jeu, col)` rétablissant, après exécution d'un appel `jouer_colonne(jeu, joueur, col)`, le jeu dans son état précédent.

Q8. Définir une fonction `initialiser_jeu(liste_colonnes)`, prenant en argument une liste de numéros de colonnes, jouées sur un plateau vide, dans l'ordre où elles sont jouées par les deux joueurs, et renvoyant le jeu dans l'état résultant.
On devra par exemple obtenir :

```
>>> L1 = [1, 0, 2, 4, 3, 5, 1, 2, 4, 3, 3, 2, 2, 4]
>>> jeu1 = initialiser_jeu(L1)
>>> affiche(jeu1)
0| _ _ _ _ _ _ _
1| _ _ _ _ _ _ _
2| _ _ 1 _ _ _ _
3|     2 1 2
4| _ 1 2 2 1 _ _
5| 2 1 1 1 2 2 _
   0 1 2 3 4 5 6
```

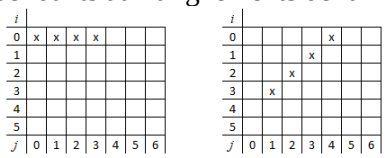
soit, par l'appel `image(jeu1)` :



On souhaite maintenant caractériser les situations dans lesquelles l'un des joueurs est gagnant.

Q9. Construire la liste, `Lalignements`, de tous les alignements de quatre cases possibles sur le plateau de jeu, sous la forme d'une liste de tuples représentant les coordonnées, (i, j) de chaque case de l'alignement.

Par exemple, on trouvera dans la liste `Lalignements`, les listes $[(0,0),(0,1),(0,2),(0,3)]$ et $[(3,1),(2,2),(1,3),(0,4)]$ correspondants aux alignements de la figure suivante.



On pourra utiliser la liste `vecteurs_deplacements` des coordonnées des quatre vecteurs de déplacements, u_1, u_2, u_3, u_4, de coordonnées respectives $(0,1)$ (déplacement horizontal), $(1,0)$ (déplacement vertical), $(1,1)$ (déplacement descendant) et $(-1,1)$ (déplacement ascendant) :

```
>>> vecteurs_deplacements = [(0, 1), (1, 0), (1, 1), (-1, 1)]
```

et la fonction suivante :

```
def translation(M: tuple, u: tuple, k: int) -> tuple:
    """renvoie les coordonnées de l'image du point M par la translation de vecteur k * u"""
    i, j = M
    ui, uj = u
    return (i + k * ui, j + k * uj)
```

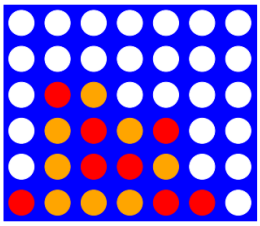
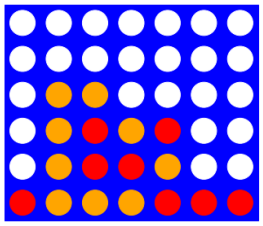
afin de générer les alignements à l'aide de compréhensions de listes.
Ainsi, les alignements donnés en exemple ci-dessus pourront être générés, respectivement, par :

```
>>> [translation((0, 0), vecteurs_deplacements[0], k) for k in range(4)]
[(0, 0), (0, 1), (0, 2), (0, 3)]
>>> [translation((3, 1), vecteurs_deplacements[3], k) for k in range(4)]
[(3, 1), (2, 2), (1, 3), (0, 4)]
```

On vérifiera que l'on obtient au total 69 alignements, dont 24 horizontaux, 21 verticaux, 12 ascendants et 12 descendants.

Q10. Construire le dictionnaire `Dalignements`, dont les clés sont les couples d'entiers des coordonnées des cases du plateau, avec, pour valeur associée à chaque clé (i,j), la liste des alignements de quatre cases passant par la case de coordonnées (i,j).
On pourra recourir à des tests `x in L`, testant si une valeur x est présente dans une liste L, de complexité $O(\text{len}(L))$ dans le pire cas.
Par exemple, pour la case en haut à gauche, `Dalignements[(0, 0)]`, devra renvoyer à la liste de trois alignements $[[(0,0),(0,1),(0,2),(0,3)],[(0,0),(1,0),(2,0),(3,0)],[(0,0),(1,1),(2,2),(3,3)]]$.

Q11. Définir une fonction `evaluer(jeu, alignement)`, renvoyant le numéro du joueur occupant les quatre cases de l'alignement si tel est le cas, et zéro sinon.
On pourra tester la fonction sur les jeux suivants, dans lesquels l'un des deux joueurs est gagnant.

Jeu 2	**Jeu 3**
	

`>>> evaluer_alignement(jeu2, \`
`    [(4, 0), (4, 1), (4, 2), (4, 3)])`
`0`
`>>> evaluer_alignement(jeu2, \`
`    [(2, 1), (3, 2), (4, 3), (5, 4)])`
`2`
	`>>> evaluer_alignement(jeu3, \`
	`    [(2, 1), (3, 1), (4, 1), (5, 1)])`
	`1`
	`>>> evaluer_alignement(jeu2, \`
	`    [(4, 1), (4, 2), (4, 3), (4, 4)])`
	`0`

Q12. Définir une fonction `teste_gagnante(jeu, joueur, coords)`, renvoyant un booléen indiquant si le joueur `joueur` est gagnant après avoir joué un coup dans la case de coordonnées `coords`.
On devra, par exemple, obtenir :

`>>> teste_gagnante(jeu2, 2, (2, 1))`
`True`
`>>> teste_gagnante(jeu2, 2, (3, 4))`
`False`
	`>>> teste_gagnante(jeu3, 1, (2, 1))`
	`True`
	`>>> teste_gagnante(jeu2, 2, (2, 1))`
	`False`

On pourra alors tester le jeu en exécutant le script présent dans le notebook.

2 Détermination d'une stratégie de jeu par l'algorithme MinMax

2.1 Définition d'une heuristique

On souhaite définir une fonction `heuristique(jeu)` permettant d'évaluer les potentialités d'un état du jeu en vue de la victoire de l'un des deux joueurs.

Q13. Construire un tableau à deux dimensions, `nb_alignements_potentiels`, de même taille que le plateau de jeu, et tel que `nb_alignements_potentiels[i][j]` donne le nombre d'alignements de quatre cases possibles, sur un plateau vide, et passant par la case de coordonnées (i, j).
On utilisera pour cela le dictionnaire `Dalignements`.
On devra obtenir les valeurs suivantes :

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

La valeur de l'heuristique pour un état du jeu donné sera prise égale à `float('inf')` si le jeu est gagnant pour le joueur 1, ou à `-float('inf')` si le jeu est gagnant pour le joueur 2, dans tous les autres cas, elle sera égale à la somme des nombres d'alignements potentiels pour chaque case occupée, comptés positivement si la case est occupée par le joueur 1 et négativement si elle est occupée par le joueur 2.

Q14. Définir une fonction `heuristique(jeu)` renvoyant la valeur de l'heuristique pour un état du jeu donné. On devra obtenir pour le jeu `jeu1`, la valeur 2, égale à la somme

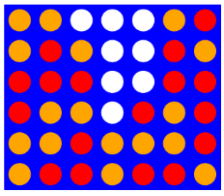
$$4 + 5 + 7 + 6 + 8 + 13 + 11 - 3 - 5 - 4 - 8 - 10 - 11 - 11.$$

L'heuristique vaudra `-float('inf')` pour le jeu 2 et `float('inf')` pour le jeu 3.
On pourra définir une fonction `signe(joueur)` renvoyant 1 ou -1 afin d'affecter les signes.

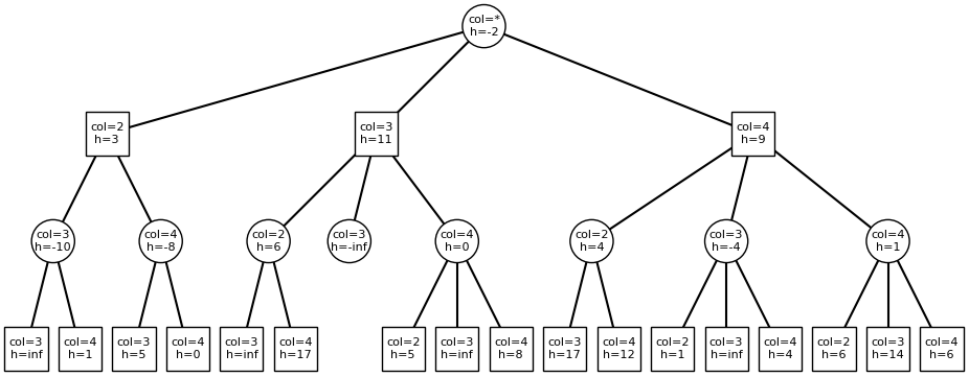
2.2 Algorithme MinMax

À partir d'un état du jeu donné, l'algorithme MinMax consiste à envisager tous les coups possibles avec un horizon de n coups, la valeur de n étant choisie arbitrairement.
Ainsi, à partir du jeu dans l'état suivant, que l'on notera jeu 4 :

```
0| 1 1         1 2
1| 1 2 1 _ _   2 1
2| 2 2 2 _ _   2 2
3| 2 1 1 _ 2   1 2
4| 1 1 2 1 1   1 2
5| 1 2 2 1 2   2 1
   0 1 2 3 4 5 6
```



L'arbre de tous les coups possibles, à un horizon de 3 coups, sachant que c'est au joueur 1 de jouer, est le suivant (chaque nœud de l'arbre a été étiqueté par le numéro de la colonne jouée – sauf pour la racine, et la valeur de l'heuristique du jeu dans l'état atteint) :



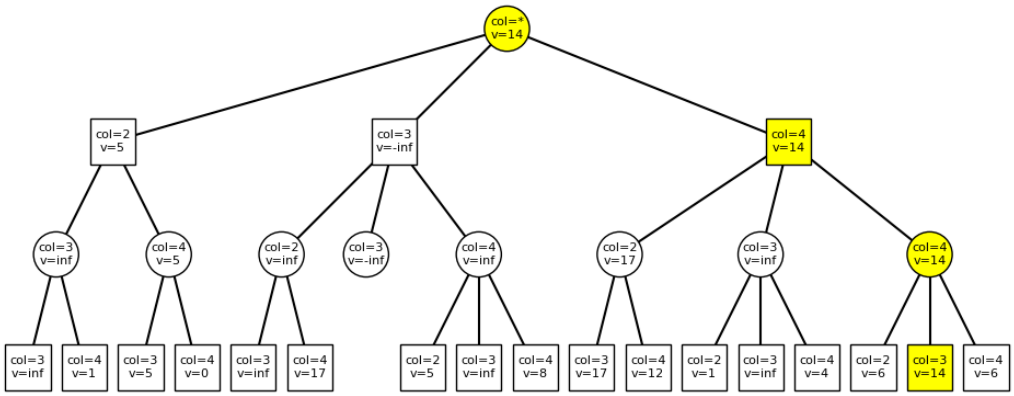
Pour le dernier coup, on considère que chaque joueur jouera le coup qui optimise l'heuristique de son point de vue :

- si c'est au joueur 1 de jouer, il choisira le coup qui maximise l'heuristique ;
- si c'est au joueur 2 de jouer, il choisira le coup qui minimise l'heuristique.

La valeur de l'heuristique du coup choisi sera affectée au nœud parent.

On procède de même pour tous les tours de jeu précédents, chaque joueur choisissant, parmi les coups suivants possibles :

- s'il s'agit du joueur 1, celui auquel la valeur maximale aura été affectée ;
- s'il s'agit du joueur 2, celui auquel la valeur minimale aura été affectée.



Ainsi, dans l'état du jeu correspondant au jeu 4, le joueur 1 choisira de jouer la colonne 4, qui lui assure, avec un horizon de trois coups, d'atteindre un état du jeu d'heuristique égale à 14 (en supposant que le joueur 2 applique la stratégie MinMax à l'horizon de deux coups).

Afin d'appliquer la stratégie MinMax, on va écrire deux fonctions mutuellement récursives, l'une, `maximin(jeu, n)`, destinée au joueur 1, et qui renvoie la colonne assurant au joueur 1 d'atteindre un état du jeu d'heuristique maximale après n coups, et la valeur de cette heuristique, l'autre, `minimax(jeu, n)`, destinée au joueur 2, et qui renvoie la colonne assurant au joueur 2 d'atteindre un état du jeu d'heuristique maximale après n coups, et la valeur de cette heuristique.

Q15. Définir les fonctions `maximin(jeu, n)` et `minimax(jeu, n)`. Pour $n > 0$, on créera, pour chaque colonne jouable, une copie du jeu dans laquelle on jouera la colonne, et on sélectionnera parmi ces colonnes, celle qui amène à l'horizon $n - 1$, l'heuristique – calculée avec la fonction de l'adversaire, de plus grande valeur si c'est au joueur 1 de jouer, celle de plus petite valeur si c'est au joueur 2 de jouer. La fonction renverra ce numéro de colonne, ainsi que la valeur de l'heuristique qu'il est possible d'atteindre. Si une colonne est gagnante pour le joueur, on interrompra la recherche et on renverra le numéro de la colonne et la valeur infinie de l'heuristique correspondante. Pour $n = 0$ ou si le plateau est rempli, les deux fonctions renverront la valeur `None` et l'heuristique du jeu.
On pourra compléter le squelette de fonction suivant, et son correspondant pour la fonction `minimax` :

```
def maximin(jeu, n):
    colonnes_possibles = ...
    if ... :
        return None, heuristique(jeu)
    ## recherche de l'heuristique maximale à l'horizon de n coups
    colmax, hmax  = None, -float('inf')
    for col in ...:
        jeu_suivant = copie(jeu)
        coords = jouer_colonne(...)
        if ...:
            return col, float('inf')
        eval_minimax = minimax(...)[1]
        if eval_minimax ...:
            ...
    return colmax, hmax
```

Pour les jeux 1 et 4, on pourra vérifier que `maximin` renvoie les valeurs suivantes pour ces différentes valeurs de l'horizon :

```
>>> maximin(jeu1, 1)
(3, 15)
>>> maximin(jeu1, 2)
(3, 4)
>>> maximin(jeu1, 3)
(3, 14)
```

```
>>> maximin(jeu4, 1)
(3, 11)
>>> maximin(jeu4, 2)
(4, -4)
>>> maximin(jeu4, 3)
(4, 14)
```

On pourra alors tester le jeu contre l'ordinateur en exécutant le script présent dans le notebook.

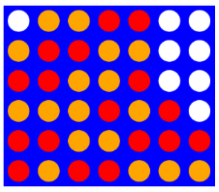
2.3 Élagage alpha-béta

La complexité de l'algorithme MinMax est, dans le pire cas, de l'ordre de 7^n (à chaque fois que l'horizon est repoussé de 1, le nombre de coups à examiner est multiplié par 7 si toutes les colonnes sont jouables).
Or, il est possible de limiter le nombre de possibilités à examiner, compte-tenu des résultats déjà obtenus. Pour cela on peut considérer, le plus grand maximum déjà trouvé pour le joueur 1, noté `alpha`, et le plus petit minimum déjà trouvé pour le joueur 2, noté `beta`.
Ayant connaissance de ces valeurs, la recherche d'un maximum pour le joueur 1 lors d'un appel à `maximin(jeu, n)`, se fait par des appels à `minimax(jeu, n - 1)`, et si `minimax` renvoie une valeur supérieure ou égale à `beta`, alors le maximum qui sera renvoyé sera à coup sûr supérieur au minimum déjà trouvé pour le joueur 2 et sera donc rejetée par la fonction `minimax(jeu, n + 1)`, lorsque sa valeur lui sera renvoyée, aussi la recherche de maximum peut être interrompue.
De même pour un appel à `minimax(jeu, n)` : la recherche de minimum peut être interrompue au premier appel à `maximin(jeu, n -1)` qui renvoie une valeur inférieure ou égale à `alpha`, car elle sera rejetée par la fonction appelante `maximin(jeu, n + 1)`.
Dans les deux cas, s'il n'y a eu aucun appel antécédent à les valeurs de `alpha` et `beta` seront prises égales, respectivement, à `-float('inf')` et `float('inf')`, indiquant qu'aucun maximum n'a encore été trouvé pour le joueur 1, respectivement, aucun minimum pour le joueur 2.
Afin de coller au déroulement du jeu, si lors d'un appel à `maximin(jeu, n)`, on obtient une valeur renvoyée par `minimax(jeu, n - 1)` inférieure à `beta`, on actualisera la valeur de `beta` (plus petit minimum déjà trouvé pour le joueur 2).
De même, si lors d'un appel à `minimaxi(jeu, n)`, on obtient une valeur renvoyée par `maximin(jeu, n - 1)` inférieure à `alpha`, on actualisera la valeur de `alpha` (plus grand maximum déjà trouvé pour le joueur 1).
En conséquence, les valeurs de `alpha` et `beta`, au moment d'un appel, seront toujours telles que

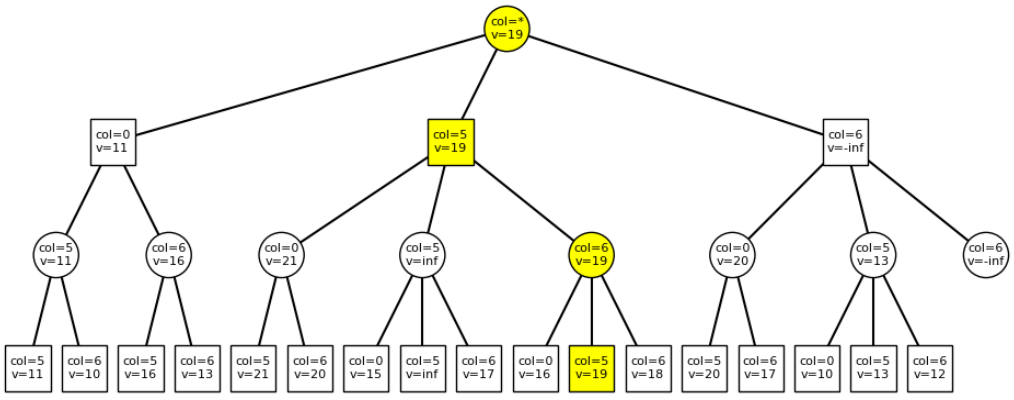
`alpha` < `beta`.

À titre d'exemple, on peut considérer le jeu suivant, noté jeu 5 :

```
jeu5 = {'nbcoups': 34, 'hauteurs': [5, 6, 6, 6, 6, 3, 2],
        'plateau': [[0, 1, 1, 2, 2, 0, 0],
                    [1, 2, 2, 1, 1, 0, 0],
                    [2, 2, 1, 1, 2, 0, 0],
                    [1, 1, 1, 2, 1, 2, 0],
                    [2, 2, 1, 1, 2, 2, 2],
                    [2, 1, 2, 2, 1, 1, 1]]}
```



dont l'exploration à l'aide de l'algorithme MinMax est décrite par l'arbre suivant.



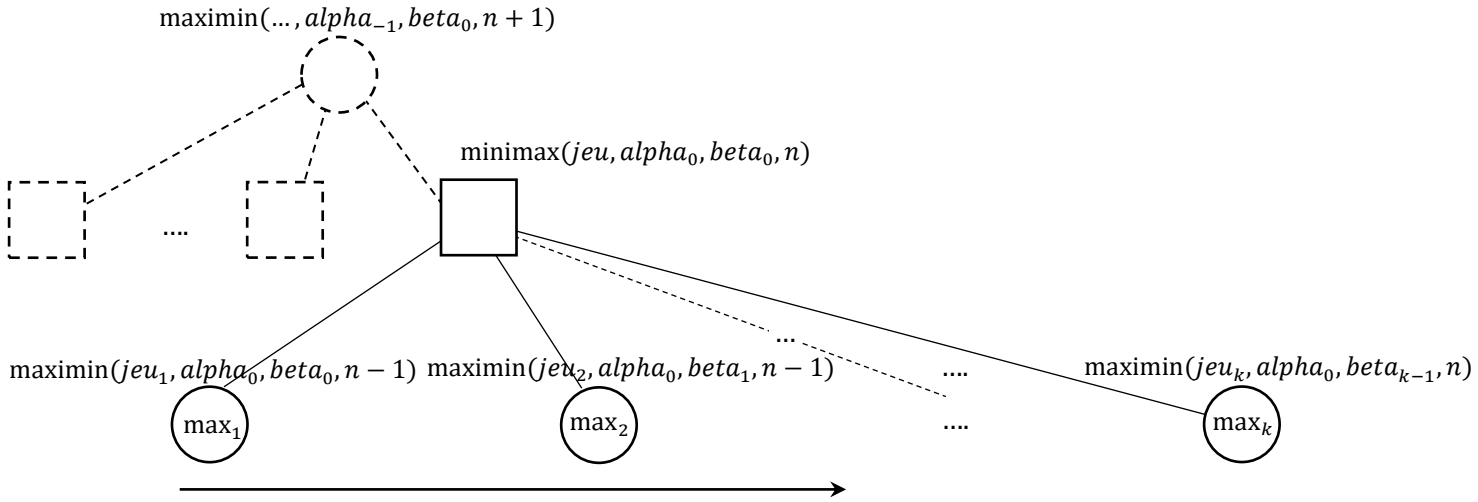
Q16. Anticiper les nœuds de cet arbre qui ne seront pas explorés si on met en œuvre un élagage alpha-béta, en s'aidant au besoin des annexes 1 et 2.

On souhaite modifier les fonctions `maximin` et `minimax` de sorte à procéder à l'élagage. Afin de tenir compte des remarques précédentes, on transmettra lors de chaque appel à l'une ou l'autre de ces deux fonctions, au travers de deux paramètres additionnels `alpha` et `beta`, respectivement, les valeurs des maximum et minimum déjà trouvés lors des explorations précédentes.

Q17. Proposer une implémentation pour les deux fonctions modifiées, que l'on nommera `maximin_alpha_beta(jeu, alpha, beta, n)` et `minimax_alpha_beta(jeu, alpha, beta, n)`.
On pourra alors tester le jeu contre l'ordinateur en exécutant le script présent dans le notebook.

ANNEXE 1

Élagage alpha



Recherche de minimum : la valeur de béta diminue au fil de la recherche (ou reste constante)

La recherche de minimum s'interrompt au premier maximum renvoyé, $\max_k$, $k \geq 1$, qui est inférieur ou égal à $alpha_0$.

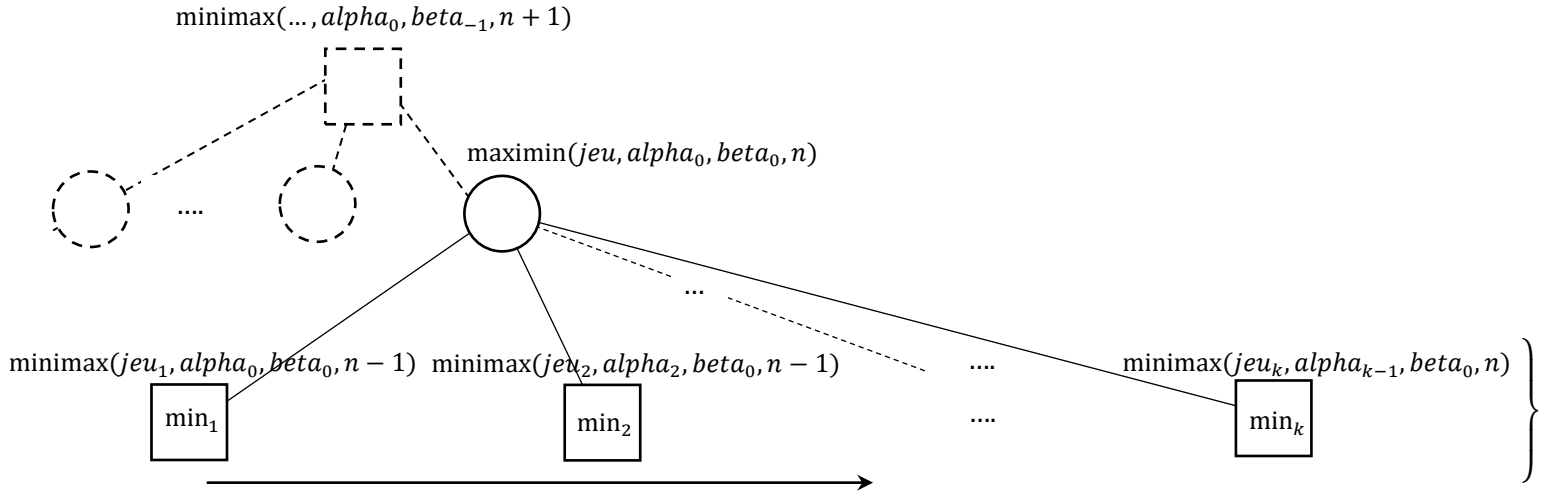
Si la situation advient, ce sera cette valeur, $\max_k$, qui sera renvoyée au titre de minimum, mais elle ne sera pas prise en compte dans le calcul de maximum de l'appel maximin(… , $alpha_{-1}, beta_0, n+1$) s'il existe.

- $alpha_0, beta_0$ sont, respectivement, les plus grand maximum et plus petit minimum déjà trouvés au moment de l'appel à maximin($jeu, alpha_0, beta_0, n$), soit lors d'appels de niveaux supérieurs, pour $beta_0$, soit, pour, $alpha_0$, aussi lors d'éventuels appels précédents à minimax initiés par l'appel maximin(… , $alpha_{-1}, beta_0, n+1$), auquel cas on aura $alpha_0 < alpha_{-1}$.

- lors du $k^{\text{ème}}$ appel à maximin , $k > 0$, $beta_{k-1}$ est le minimum des valeurs $beta_0$, $\max_1$, $\max_2$, …, $\max_{k-1}$.

ANNEXE 2

Élagage béta



Recherche de maximum : la valeur de alpha augmente au fil de la recherche (ou reste constante)

La recherche de maximum s'interrompt au premier minimum renvoyé, $\min_k$, $k \geq 1$, qui est supérieur ou égal à $beta_0$.

Si la situation advient, ce sera cette valeur, $\min_k$, qui sera renvoyée au titre de maximum, mais elle ne sera pas prise en compte dans le calcul de minimum de l'appel minimax(… , $alpha_0, beta_{-1}, n+1$) s'il existe.

- $alpha_0, beta_0$ sont, respectivement, les plus grand maximum et plus petit minimum déjà trouvés au moment de l'appel maximin($jeu, alpha_0, beta_0, n$), soit lors d'appels de niveaux supérieurs, pour $alpha_0$, soit, pour, $beta_0$, aussi lors d'éventuels appels précédents à maximin initiés par l'appel minimax(… , $alpha_0, beta_{-1}, n+1$), auquel cas on aura $beta_0 < beta_{-1}$.

- lors du $k^{\text{ème}}$ appel à minimax , $k > 0$, $alpha_{k-1}$ est le maximum des valeurs $alpha_0$, $\min_1$, $\min_2$, …, $\min_{k-1}$.