

Théorie des Jeux

Contexte

La théorie des jeux est une branche des mathématiques qui étudie les interactions stratégiques entre des agents rationnels. Plus précisément, elle analyse les situations où le succès d'un individu dépend des choix des autres. Un des aspects fascinants de cette théorie concerne les jeux d'accessibilité à deux joueurs sur un graphe. Ces jeux sont utilisés pour modéliser et résoudre des problèmes dans divers domaines tels que l'informatique, l'économie, et la biologie.

Le programme se limite aux jeux d'accessibilité à deux joueurs sur un graphe. Les joueurs se déplacent de sommet en sommet en suivant les arêtes, avec l'objectif d'atteindre ou d'éviter certaines positions spécifiques. Un sommet est aussi appelé une **position**. La notion de graphe est suffisamment riche pour englober une grande variété de jeux et de contextes compétitifs. En effet, les graphes peuvent modéliser des réseaux sociaux, des circuits électroniques, des routes de transport, et bien d'autres systèmes complexes où les interactions entre les éléments peuvent être représentées par des connexions. Cette flexibilité fait des graphes un outil puissant pour analyser les stratégies et les dynamiques dans des environnements divers et compétitifs.

Une **stratégie** est un plan d'action complet pour un joueur, définissant les mouvements à effectuer en fonction de chaque situation rencontrée. En d'autres termes, c'est une règle prédéfinie qui indique comment le joueur doit agir à chaque étape du jeu pour atteindre son objectif.

Une **stratégie gagnante** est une stratégie qui garantit à un joueur de gagner, quel que soit le comportement de l'adversaire. Dans le contexte des jeux d'accessibilité, cela signifie qu'un joueur peut toujours atteindre une position cible, malgré les efforts de l'autre joueur pour l'en empêcher. Une **position gagnante** est un sommet à partir duquel un joueur peut garantir une victoire en suivant une stratégie gagnante. La détermination des positions gagnantes est cruciale pour comprendre la dynamique du jeu et développer des stratégies efficaces.

Les **attracteurs** sont des ensembles de positions vers lesquels un joueur peut forcer le jeu à évoluer, indépendamment des actions de l'adversaire. Le calcul des attracteurs permet d'identifier les positions gagnantes en déterminant les sommets d'où un joueur peut toujours amener le jeu vers une position favorable.

La construction de stratégies gagnantes repose sur l'analyse des attracteurs et des positions gagnantes. Une fois ces éléments identifiés, un joueur peut élaborer un plan détaillé pour atteindre les positions cibles et contourner les obstacles posés par l'adversaire.

Les jeux d'accessibilité à deux joueurs sur un graphe offrent un cadre puissant pour analyser des situations stratégiques complexes. La compréhension des concepts de stratégie, stratégie gagnante, position gagnante, et attracteurs est essentielle pour modéliser et résoudre ces jeux. À travers ce cours, nous explorerons en profondeur ces notions, en apprenant à déterminer les positions gagnantes et à construire des stratégies gagnantes, enrichissant ainsi notre compréhension des interactions stratégiques et des décisions optimales dans des contextes compétitifs.

1 Premier Exemple : Jeu de Nim avec 24 allumettes

1. On dispose de 24 allumettes alignées.
2. Deux joueurs jouent à tour de rôle.
3. À chaque tour, un joueur peut prendre soit une, soit deux allumettes.
4. Le joueur qui prend la (ou les) dernière(s) allumette(s) perd la partie.

Exemple

Pour illustrer ce jeu, voici un exemple de déroulement complet d'une partie :

Début du Jeu : 24 allumettes

Tour de Joueur A

Joueur A commence et prend 1 allumette.
Il reste : 23 allumettes

Tour de Joueur B

Joueur B prend 2 allumettes.
Il reste : 21 allumettes

Tour du Joueur A

Joueur A prend 1 allumette.
Il reste : 20 allumettes

Tour de Joueur B

Joueur B prend 2 allumettes.
Il reste : 18 allumettes

Tour du Joueur A

Joueur A prend 1 allumette.
Il reste : 17 allumettes

Tour de Joueur B

Joueur B prend 2 allumettes.
Il reste : 15 allumettes

Tour du Joueur A

Joueur A prend 1 allumette.
Il reste : 14 allumettes

Tour de Joueur B

Joueur B prend 2 allumettes.
Il reste : 12 allumettes

Tour du Joueur A

Joueur A prend 1 allumette.
Il reste : 11 allumettes

Tour de Joueur B

Joueur B prend 2 allumettes.
Il reste : 9 allumettes

Tour du Joueur A

Joueur A prend 1 allumette.
Il reste : 8 allumettes

Tour de Joueur B

Joueur B prend 2 allumettes.
Il reste : 6 allumettes

Tour du Joueur A

Joueur A prend 1 allumette.
Il reste : 5 allumettes

Tour de Joueur B

Joueur B prend 2 allumettes.
Il reste : 3 allumettes

Tour du Joueur A

Joueur A prend 1 allumette.
Il reste : 2 allumettes

Tour de Joueur B

Joueur B prend 2 allumettes.
Il reste : 0 allumettes
Joueur B perd car il a pris la dernière allumette.



Réflexion sur les derniers coups

1. Le joueur B a évidemment **mal joué** au dernier coup. Quel coup devait-il jouer pour obtenir à coup sûr la victoire? **Réponse**

Tour de Joueur B

Joueur B prend 1 allumette.
Il reste : 1 allumette

Tour du Joueur A

Joueur A prend 1 allumette.
Il reste : 0 allumettes Joueur A perd car il a pris la dernière allumette.

Le coup correct pour Joueur B dans la situation actuelle (au dernier coup où il reste 2 allumettes) aurait été de prendre 1 allumette, laissant 1 allumette à A.

2. Mais A pouvait anticiper, et ne devait pas laisser cette possibilité à B! Quel coup aurait alors été judicieux pour A? **Réponse** Pour garantir que B soit en position perdante et éviter que B ne puisse gagner, A doit prendre 2 allumettes lorsqu'il en reste 3. Cela laisse 1 allumette à B, qui est une position perdante pour lui.

Tour du Joueur A

Joueur A prend 2 allumettes.
Il reste : 1 allumette

Tour de Joueur B

Joueur B prend 1 allumette.
Il reste : 0 allumettes
Joueur B perd car il a pris la dernière allumette.

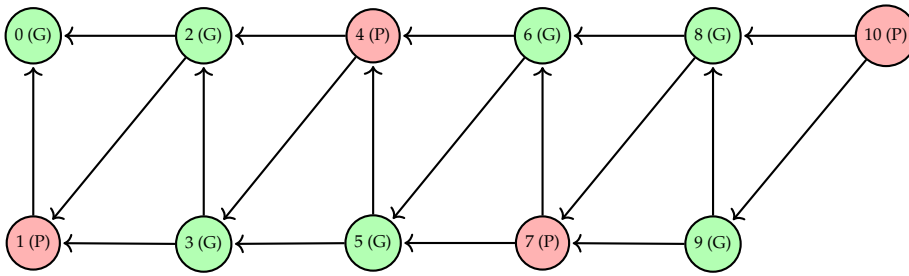
3. En fait, de la part de B, laisser 3 allumettes à A était un très mauvais coup! En effet il suffit alors à A de retirer deux allumettes. Comment B aurait-il pu jouer, lorsqu'il a hérité de 5 allumettes? **Réponse** B prend une allumette, en laissant ainsi 4 à A. Si A retire une allumette, B se retrouve avec 3 allumettes et l'on a vu que cette position permettait d'obtenir la victoire. Si A prend 2 allumettes, alors B en reçoit 2 et il lui suffit d'en retirer une. Ainsi, à partir de 5 allumettes, B pouvait forcer la victoire.
4. Mais alors, de la part de A, laisser 5 allumettes à B était un mauvais coup. Pouvait-il faire mieux? **Réponse** En fait si A retire une allumette il en laisse 5, mais s'il en retire 2 il en laisse 4. On a déjà vu que cette position est perdante. Ainsi A aurait mieux joué en retirant deux allumettes pour en laisser 4 à B.
5. Il ne fallait donc PAS que B laisse 6 allumettes à A. Et on pourrait remonter encore longtemps...



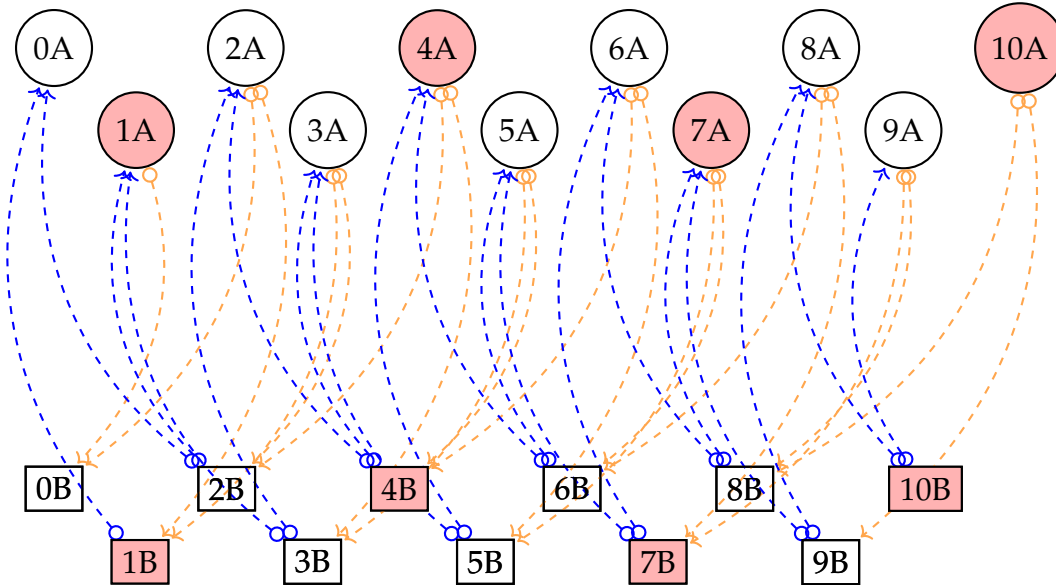
Remarque

On comprend à travers ces exemples que :

1. Certaines configurations sont définitivement perdantes, comme celle à 1 allumette.
2. Certaines configurations sont **gagnantes**, au sens où il **existe** un coup qui permet de laisser à l'autre joueur une position défavorable. Ici, 2 et 3 sont des position gagnantes : en ôtant respectivement 1 ou 2 allumettes, on laisse à l'adversaire la position perdante 1. Par contre, 4 est une position perdante car un joueur ne peut donner que des positions gagnantes à l'autre joueur (les positions 2 et 3). **Tous les coups** mènent à des positions gagnantes, que l'on donne à l'adversaire. De même, 5 et 6 sont gagnantes, car **il existe** un coup permettant de laisser la position gagnante 4.
3. La configuration 7 est _____, les positions _____ sont gagnantes.
4. On peut représenter sur un graphe orienté toutes les positions de 24 à 0. Celui qui hérite de la position 0 gagne, celui qui hérite de la position 1 perd. Une flèche indique qu'il existe un coup permettant de passer d'une position à une autre. On colorie alors récursivement les positions gagnantes et perdantes.
Le graphe suivant représente les positions du jeu avec leurs couleurs associées.



Disposer d'un graphe biparti **en dédoublant les sommets** peut être plus intéressant et lisible :



▲ DÉFINITION 1

Un graphe est **biparti** si

1. l'ensemble S de ses sommets est partitionné en deux sous-ensembles S_A et S_B tels que $S = S_A \cup S_B$ et $S_A \cap S_B = \emptyset$,
2. les arêtes ne peuvent relier qu'un noeud de S_A à un noeud de S_B , et réciproquement.

2 Cadre général

Dans un jeu, des joueurs, à partir d'une situation donnée, prennent des décisions à tour de rôle parmi un ensemble fini de décisions possibles, chacune menant à une nouvelle situation. Le programme se limite à des jeux :

- **à deux joueurs jouant à tour de rôle** ;
- **à information complète**, c'est-à-dire chaque joueur connaît l'ensemble de la situation ;
- **sans mémoire**, c'est-à-dire la décision est prise seulement en fonction de la situation actuelle, pas des situations passées ;
- **sans hasard**.

Tous ces jeux peuvent être **modélisés** par des graphes orientés finis : chaque sommet est une situation (on parle aussi de position) et chaque arête correspond à une décision, un coup possible. On peut alors voir le jeu comme le déplacement d'un jeton sur ce graphe : on démarre sur un sommet donné et, à tour de rôle, les joueurs déplacent le jeton en suivant des arêtes. Le jeu se termine lorsque le jeton arrive à un sommet gagnant pour A, gagnant pour B, ou signifiant l'égalité. Il est important que le graphe ne possède pas de cycle pour que le jeu puisse se terminer.

On s'intéressera uniquement aux jeux d'accessibilité : pour gagner, chaque joueur a pour but d'atteindre un ou plusieurs sommets particuliers.

On peut par exemple penser au morpion, au puissance 4, aux dames, aux échecs, au go, etc.

▲ DÉFINITION 2

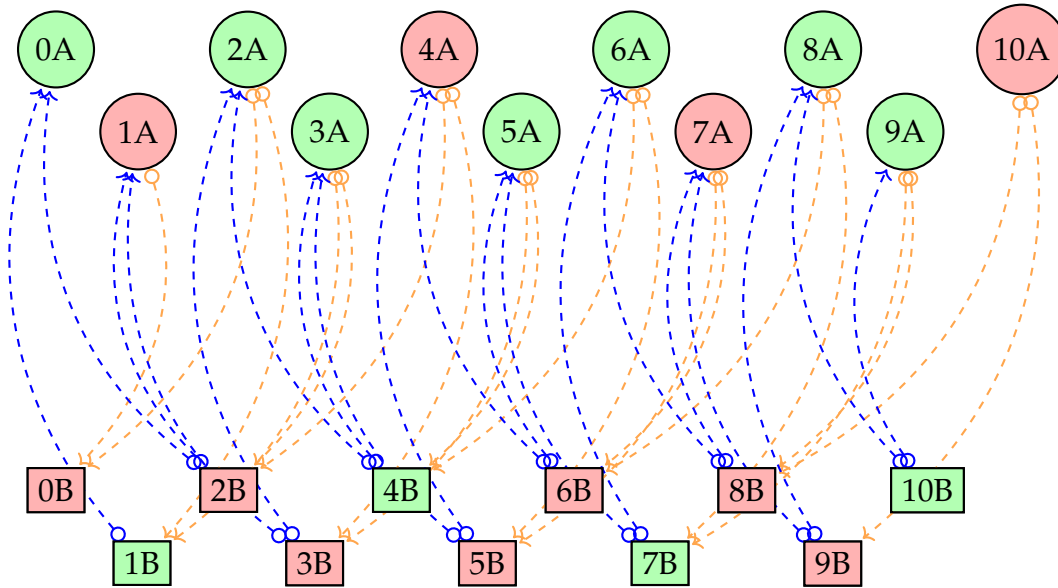
Plaçons(-nous dans la situation du graphe biparti pour représenter le jeu. On définit l'attracteur du joueur A ainsi.

1. On note A_0 l'ensemble des configurations pour lesquelles A est déclaré gagnant.
2. Si A_n est bien défini, on note A_{n+1} l'ensemble des sommets s tels que l'une des propriétés est bien vérifiée :
 - $s \in A_n$;
 - $s \in S_A$ et s a pour successeur au moins un sommet de A_n ;
 - $s \in S_B$ et tout successeur de s est un sommet de A_n .

On a donc $A_{n+1} := A_n \cup \{s \in S_A \mid \exists t \in A_n s \rightarrow t\} \cup \{s \in S_B \mid \forall t \in S s \rightarrow t \Rightarrow t \in A_n\}$.

On pose $A_\infty := \bigcup_{n \in \mathbb{N}} A_n$ la réunion croissante des parties A_n , c'est l'**attracteur de A** .

On définit de même l'attracteur de B .



On représente ici

en vert l'attracteur de A et en rouge celui de B . On constate que ces deux attracteurs sont disjoints. Ici ils partitionnent l'ensemble des sommets, ce qui traduit en particulier l'absence de partie nulle.

→ PROPOSITION 1

Le sommet s appartient à l'attracteur de A si, et seulement si, A a une stratégie gagnante depuis le sommet s .

De même s appartient à l'attracteur de B si, et seulement si, B a une stratégie gagnante depuis le sommet s .

☞ COROLLAIRE 1

Pour gagner, il « suffit » de :

1. Déterminer les attracteurs du jeu
2.
 - Si le sommet est dans votre attracteur, jouer de façon à rester dans l'attracteur
 - Sinon, joueur au hasard.



Attention

Le mot « suffit » est réellement une condition suffisante au sens mathématique du terme, mais les guillemets sont là pour signifier que cela n'a en général pas de contrepartie pratique. Pour les jeux réels auxquels on joue, le graphe des positions dispose d'un très grand nombre de sommets, et il n'est pas question de déterminer l'attracteur.

Ce résultat est donc surtout théorique. Une de ses interprétations est qu'avec une puissance de calcul infinie, il est possible de jouer parfaitement à tous les jeux à deux joueurs sans mémoire et à information totale tels que Go, Dames, Echecs, Puissance 4, Hex...

3 Heuristique et exemple du Tic Tac Toe (TD 1)

Il n'est pas possible d'explorer chaque configuration pour déterminer l'attracteur; on remplace cette exploration exhaustive par une « note » de la position. Une note élevée correspond à une position très probablement intéressante, une note basse à une position défavorable (et donc favorable à l'adversaire). Cette note s'appelle une **heuristique**. Une heuristique est donc une fonction h de S vers \mathbb{R} .

Chaque joueur tente de maximiser ou minimiser h . Supposons par exemple que c'est à A de jouer. Parmi différentes positions possibles, A choisit le coup qui donne la position d'heuristique maximale. De même, B cherchera la position d'heuristique minimale (s'il utilise la même heuristique que A). Il est bien sûr possible que A et B n'aient pas la même heuristique.



Rappel des règles du Tic Tac Toe

Le Tic Tac Toe, aussi appelé morpion, est un jeu simple qui se joue à deux joueurs sur une grille de 3x3. Voici les règles :

1. **Objectif** : Chaque joueur essaie d'aligner trois de ses symboles (X ou O) horizontalement, verticalement ou en diagonale.
2. **Mise en place** : La grille est initialement vide. Les joueurs choisissent leurs symboles, l'un joue avec 'X' et l'autre avec 'O'.
3. **Tour de jeu** : Les joueurs jouent à tour de rôle. Le joueur avec les 'X' commence généralement.
4. **Placement** : À chaque tour, un joueur place son symbole dans une case vide de la grille.
5. **Victoire** : Un joueur gagne s'il parvient à aligner trois de ses symboles horizontalement, verticalement ou en diagonale.
6. **Match nul** : Si toutes les cases de la grille sont remplies sans qu'aucun joueur n'ait aligné trois symboles, la partie se termine par un match nul.

Exemple

Imaginons une partie entre deux joueurs, X et O :

Grille initiale

X		

Tour 1 (X)

X		
	O	

Tour 2 (O)

X	X	
	O	

Tour 3 (X)

X	X	
O		

Tour 4 (O)

X	X	X
O		

Tour 5 (X)

À ce stade, le joueur X a gagné en alignant trois 'X' horizontalement sur la première ligne.

3.1 Représentation

Une configuration d'une grille est représentée par une 3-liste de 3-listes, dont les éléments sont des chaînes de caractères 'O', 'X' ou ' '. Les deux joueurs ici ne seront pas désignés par la lettre A ou B, mais par le symbole X ou O.

Exemple

O		X
	X	
O		

La configuration : est représentée par la liste de listes de caractères :

```
[[ 'O', ' ', 'X'], [ ' ', 'X', ' '], [ 'O', ' ', ' ']]
```

Dans cette configuration c'est à X de jouer et il dispose d'un seul mouvement pour éviter la défaite lorsque ce sera au tour de O. Voyez-vous lequel ?

Exercice 1 Ecrire une fonction qui `afficher_plateau` qui reçoit en argument une configuration sous forme de liste et dessine un plateau de jeu.

Par exemple `afficher_plateau([['O', ' ', 'X'], ['X', 'X', 'O'], ['O', ' ', ' ']])` renvoie :

```
-----
| O |   | X |
-----
| X | X | O |
-----
| O |   |   |
```

3.2 Condition de Victoire

Exercice 2 La fonction `check_win` vérifie si une position `board` est gagnante pour le joueur `player` ; `player` est une chaîne de caractères 'X' ou 'O', et `board` est une position comme décrite précédemment.

```
1 def check_win(board, player):
2     # Verifier les lignes, colonnes et diagonales pour une victoire
3     for row in range(3):
4         if all([cell == player for cell in board[row]]):
5             return True
6
7     for col in range(3):
8         if all([board[row][col] == player for row in range(3)]):
9             return True
10
11    if all([board[i][i] == player for i in range(3)]) or all([board[i][2 - i] == player
12        for i in range(3)]):
13        return True
14
15    return False
```

Ce code est juste, mais certaines commandes sont hors programme en CPGE actuellement. Réécrivez cette fonction en respectant le programme de PC. Vous pourrez, au choix, écrire des sous-fonctions sans changer la fonction, ou bien changer le code dans le corps de la fonction.

3.3 Heuristiques

3.3.1 Alignement

Exercice 3 Compléter les trous dans cette fonction, qui a pour objectif de déterminer la valeur d'une ligne du point de vue du joueur X.

```
1 def evaluate_line(cell1, cell2, cell3, player):
2     score = 0
3
4
5     opponent = 'O' if player == 'X' else 'X'
6
7     # Premier cas : toutes les cellules sont du joueur
8     if cell1 == cell2 == cell3 == player:
9         score = 1000
10    # Deuxieme cas : deux cellules sont du joueur et une est vide
11    elif cell1 == cell2 == player and cell3 == ' ':
12        score = 10
13    elif cell1 == cell3 == player and cell2 == ' ':
14        score = 10
15    elif A COMPLETER:
16        score = 10
17    # Troisieme cas : une cellule est du joueur et les autres sont vides
18    elif cell1 == player and cell2 == cell3 == ' ':
19        score = 1
20    elif A COMPLETER:
21        score = 1
22    elif A COMPLETER:
23        score = 1
24
25    # Deduire les points si l'adversaire est en position similaire
26    if A COMPLETER:
27        score = -1000
28    elif A COMPLETER:
29        score = -10
30    elif cell1 == cell3 == opponent and cell2 == ' ':
31        score = -10
32    elif cell2 == cell3 == opponent and cell1 == ' ':
33        score = A COMPLETER
34    elif A COMPLETER:
35        score = -1
36    elif A COMPLETER:
37        score = A COMPLETER
38    elif A COMPLETER:
39        score = A COMPLETER:
```

```
40 return score
```

Aide pour comprendre la cahier des charges. Prenons la configuration :

```
1 >>> board = [
2     ['X', 'O', 'X'],
3     [' ', 'X', 'O'],
4     [' ', ' ', 'X']
5 ]
6 >>> afficher_plateau(board)
7 -----
8 | X | O | X |
9 -----
10 |   | X | O |
11 -----
12 |   |   | X |
13 -----
14 >>> evaluate_line(board[2][0], board[2][1], board[2][2], 'X')
15 1
```

Dans le tableau suivant, la fonction `evaluate_line` évalue chaque ligne, colonne et diagonale pour le joueur 'X'.

Type	Indices	Configuration	Valeur	Commande
Ligne	(0,0)-(0,2)	X, O, X	0	<code>evaluate_line(board[0][0], board[0][1], board[0][2], 'X')</code>
Ligne	(1,0)-(1,2)	, X, O	0	<code>evaluate_line(board[1][0], board[1][1], board[1][2], 'X')</code>
Ligne	(2,0)-(2,2)	, , X	1	<code>evaluate_line(board[2][0], board[2][1], board[2][2], 'X')</code>
Colonne	(0,0)-(2,0)	X, ,	1	<code>evaluate_line(board[0][0], board[1][0], board[2][0], 'X')</code>
Col	(0,1)-(2,1)	O, X,	0	<code>evaluate_line(board[0][1], board[1][1], board[2][1], 'X')</code>
Col	(0,2)-(2,2)	X, O, X	0	<code>evaluate_line(board[0][2], board[1][2], board[2][2], 'X')</code>
Diag	(0,0)-(2,2)	X, X, X	1000	<code>evaluate_line(board[0][0], board[1][1], board[2][2], 'X')</code>
Diag	(0,2)-(2,0)	X, X,	10	<code>evaluate_line(board[0][2], board[1][1], board[2][0], 'X')</code>



Exercice 4 Voici une fonction d'heuristique qui utilise les alignements :

```
1 def heuristique_alignement(board, player):
2     score = 0
3
4     # Verifie les lignes
5     for row in range(3):
6         score += evaluate_line(board[row][0], board[row][1], board[row][2], player)
7
8     # Verifie les colonnes
9     for col in range(3):
10        score += evaluate_line(board[0][col], board[1][col], board[2][col], player)
11
12    # Verifie les diagonales
13    score += evaluate_line(board[0][0], board[1][1], board[2][2], player)
14    score += evaluate_line(board[0][2], board[1][1], board[2][0], player)
15
16    return score
```

Que se passe-t-il si on exécute `heuristique_alignement(['X', 'O', 'X'], [' ', 'X', 'O'], [' ', ' ', 'X'], 'X')` ?

Proposer une autre fonction d'heuristique, appelée `heuristique_absolue` qui évalue la force d'une position en fonction des valeurs des cases occupées indépendamment du contexte. On attribuera une valeur à chaque case en fonction de son avantage stratégique, et on sommera ou soustraira ces valeurs.

3.3.2 Choix machine du meilleur coup

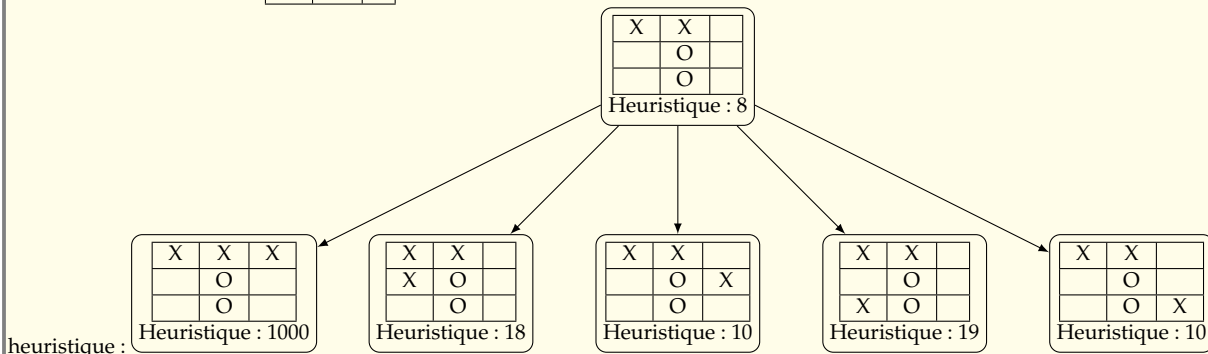
La notion d'heuristique permet à la machine de choisir le meilleur coup parmi un ensemble de coups possibles.

Exemple

Partant de la configuration

X	X	
	O	
	O	

, lorsque c'est à X de jouer, l'ordinateur considère **toutes** les positions accessibles et calcule leur



Il retient alors une position d'heuristique maximale (ici, il s'agit de

X	X	X
	O	
	O	

) et sait donc quel coup jouer.

Exercice 5 On programme ainsi la fonction `meilleur_coup`.

```

FONCTION meilleur_coup(board, player, heuristique)
    INITIALISER best_score = -infinity
    INITIALISER best_move = None

    POUR chaque ligne de 0 à 2
        POUR chaque colonne de 0 à 2
            SI board[ligne][colonne] est vide ( ' ')
                PLACER le symbole player dans board[ligne][colonne]
                CALCULER score en utilisant heuristique(board, player)
                REMETTRE board[ligne][colonne] à vide ( ' ')

                SI score est supérieur à best_score
                    METTRE À JOUR best_score avec score
                    METTRE À JOUR best_move avec (ligne, colonne)

    RETOURNER best_move
    
```

On remarquera que `best_move` est, en sortie, ou bien `None` si aucune case n'était disponible, ou bien un couple (ligne, colonne) qui maximise le score calculé par la fonction heuristique. Ecrire cette fonction en Python.

Exercice 6 On dispose de deux heuristiques, `heuristique_alignement` et `heuristique_absolue`. Donner un exemple de plateau pour lequel les deux heuristiques préconisent un meilleur coup différent.

3.4 Partie complète

Exercice 7 Voici une fonction qui permet d'organiser une **boucle de jeu**. Au cours du jeu, le joueur humain doit choisir les cases. De plus, le joueur ordinateur est paramétré de façon fermée par l'humain : il joue en premier ou deuxième, avec l'une des deux heuristiques.

```

1 def jouer_tic_tac_toe_humain_vs_ordi():
2     # Choisir qui commence
3     first_player = input("Qui commence? (H pour humain, O pour ordinateur):
4     ").strip().upper()
5     while first_player not in ['H', 'O']:
6         first_player = input("Entree invalide. Choisissez H pour humain ou O pour
7         ordinateur: ").strip().upper()
8
9     # Choisir l'heuristique
10    print("Choisissez l'heuristique:")
11    print("1. Heuristique alignement")
12    print("2. Heuristique absolue")
13    heuristique_choice = input("Entrez le numero de l'heuristique (1 ou 2): ").strip()
14    while heuristique_choice not in ['1', '2']:
    
```


```

13     heuristique_choice = input("Entree invalide. Entrez 1 ou 2: ").strip()
14
15     heuristique = heuristique_alignement if heuristique_choice == '1' else
        heuristique_absolue
16
17     board = [[' ' for _ in range(3)] for _ in range(3)]
18     current_player = 'X' if first_player == 'H' else 'O'
19
20     while True:
21         afficher_plateau(board)
22
23         if current_player == 'X':
24             # Tour de l'humain
25             valid_input = False
26             while not valid_input:
27                 row = input("Entrez la ligne (0, 1, 2) : ")
28                 col = input("Entrez la colonne (0, 1, 2) : ")
29
30                 if row.isdigit() and col.isdigit():
31                     row = int(row)
32                     col = int(col)
33
34                     if 0 <= row <= 2 and 0 <= col <= 2:
35                         if board[row][col] == ' ':
36                             board[row][col] = 'X'
37                             valid_input = True
38                         else:
39                             print("Case deja occupee, essayez de nouveau.")
40                     else:
41                         print("Entree hors limites, veuillez entrer des nombres entre 0
                            et 2.")
42                     else:
43                         print("Entree invalide, veuillez entrer des nombres entre 0 et 2.")
44         else:
45             # Tour de l'ordinateur
46             move = meilleur_coup(board, 'O', heuristique)
47             if move:
48                 board[move[0]][move[1]] = 'O'
49             else:
50                 print("Aucun coup possible, la partie est terminee.")
51                 break
52
53         if check_win(board, current_player):
54             afficher_plateau(board)
55             print(f"Le joueur {current_player} a gagne!")
56             break
57
58         if all(cell != ' ' for row in board for cell in row):
59             afficher_plateau(board)
60             print("La partie est un match nul!")
61             break
62
63         # Changer de joueur
64         current_player = 'O' if current_player == 'X' else 'X'

```

Consigne : modifier cette fonction pour l'adapter à deux joueurs **machines**. La fonction sera une fonction d'en-tête `def jouer_tic_tac_toe(heuristique_X=heuristique_alignement, heuristique_O=heuristique_alignement)` qui déroulera le jeu et donnera le vainqueur X ou O. Par défaut X et O utiliseront la fonction heuristique d'alignement, mais toute fonction d'heuristique pourra être programmée et donnée en paramètre.

On est donc capable de gérer des **tours de jeu**. Voici une fonction `jouer_tic_tac_toe` :

 **Exercice 8** Exécuter cette fonction en faisant se combattre différentes heuristiques.

4 Minimax et heuristiques : exemple du Puissance 4 – TD

La méthode du Minimax est une méthode fondamentale en théorie des jeux et en intelligence artificielle.

Partons du constat que les heuristiques sont souvent imparfaites. Elles ne notent pas une position **au vu de ce qui peut advenir**, mais en fonction de critères reconnus comme pertinents. On cherche donc à améliorer les prédictions de cette heuristique. On se rend compte, par exemple, que le joueur A peut faire mieux que uniquement se baser sur l'évaluation heuristique. En effet, en supposant que l'adversaire B joue de manière optimale et utilise la même heuristique que A, alors B cherchera à obtenir, après le coup de A, une position d'heuristique minimale (la position d'heuristique minimale est la plus intéressante pour B, car la moins favorable

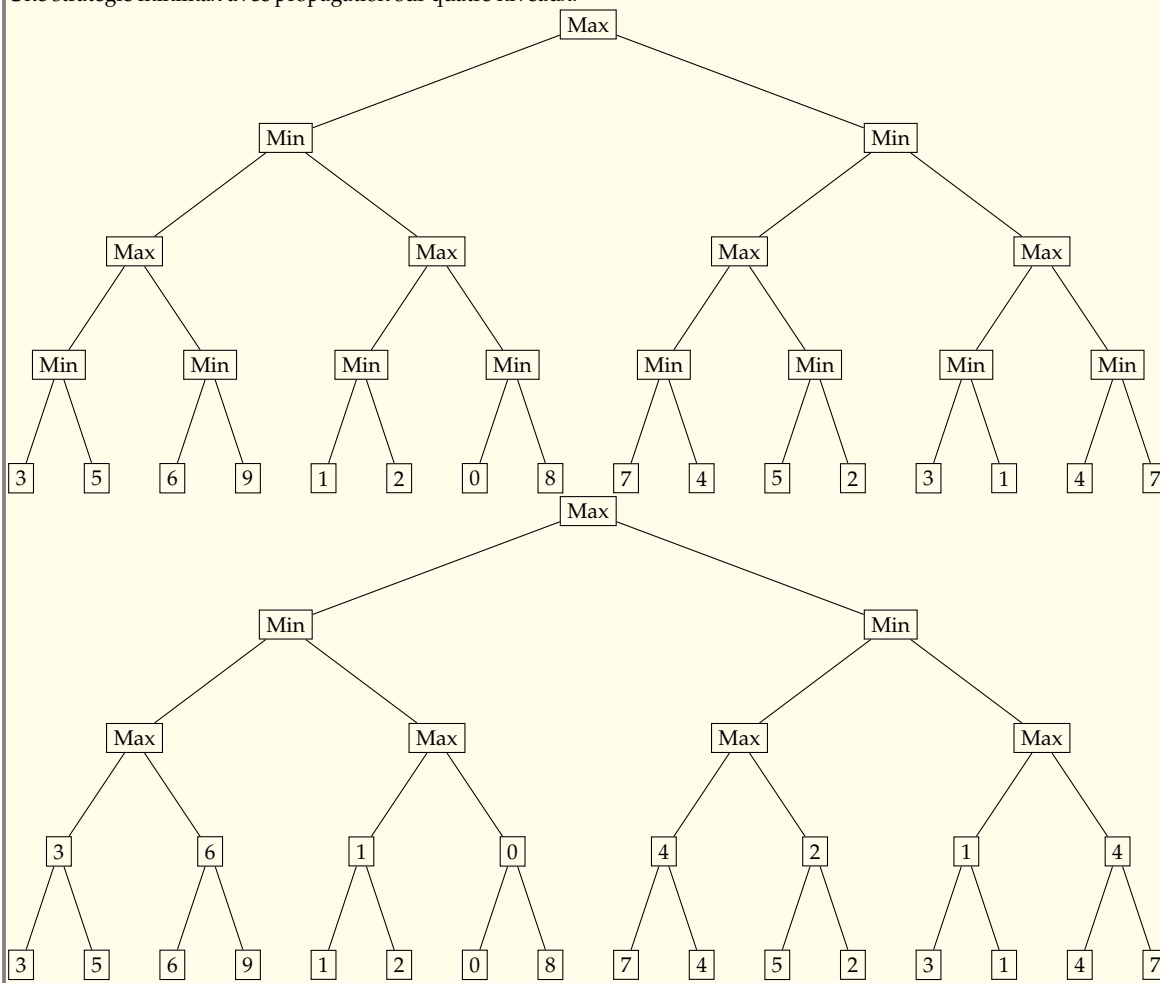
pour A). Ainsi, A peut anticiper les mouvements de B en prenant le maximum (niveau 1) des minima (niveau 2) des positions accessibles. Cette approche max-min permet de déterminer les mouvements optimaux en évaluant les résultats possibles de chaque action. Bien sûr, on peut utiliser cette méthode avec une profondeur plus grande.

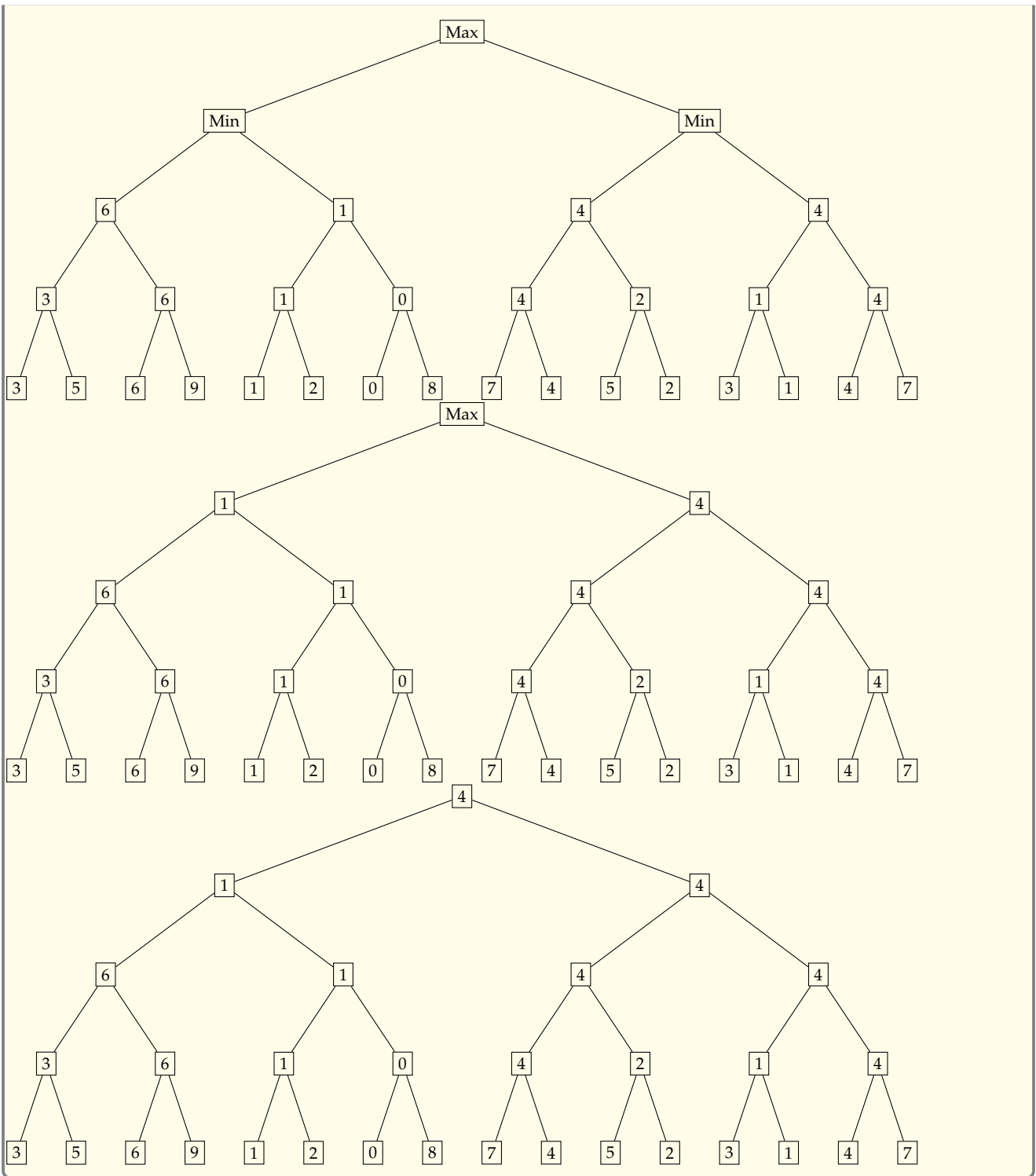
Le processus du Minimax se déroule comme suit :

1. Génération de l'Arbre de Jeu : Construire un arbre de jeu où chaque noeud représente un état du jeu et chaque branche représente un coup possible.
2. Si l'arbre est trop grand, on se limite à un certain niveau d'accessibilité : les configurations accessibles par A, ou par A puis B, ou par A puis B puis A...
3. Évaluation des Feuilles : Appliquer une fonction d'évaluation pour attribuer un score aux positions terminales (feuilles) de l'arbre.
4. Propagation des Scores : Propager les scores des feuilles vers la racine en choisissant, pour les noeuds correspondant au joueur actuel, le score maximum des successeurs (la stratégie de A sera de maximiser les gains), et pour les noeuds correspondant à l'adversaire, le score minimum des successeurs (la stratégie de B sera de minimiser les pertes).
5. Sélection du Meilleur Coup : Le joueur choisit le coup correspondant au score minimax à la racine de l'arbre.

Exemple

Une stratégie minimax avec propagation sur quatre niveaux.





Remarques

1. Cette méthode fonctionne très bien dans les **jeux à somme nulle** à deux joueurs. Dans ces jeux, les gains de l'un des joueurs sont égaux aux pertes de l'autre, ce qui signifie que les intérêts des joueurs sont strictement opposés. Dans les jeux où interviendrait une notion de coopération, une approche minimax ne fonctionne pas.
2. Le Minimax est souvent optimisé par l'algorithme d'élagage alpha-bêta (Hors programme), qui réduit le nombre de nœuds à évaluer en éliminant les branches qui ne peuvent pas influencer la décision finale. Cette optimisation est essentielle pour traiter des jeux complexes avec de grands arbres de jeu, comme les échecs ou le go.

Solution 3.3.1 Évaluation de la fonction `heuristique_alignement`

Pour la configuration du plateau :

$$\begin{bmatrix} X & O & X \\ & X & O \\ & & X \end{bmatrix}$$

et `player = 'X'`, nous allons évaluer chaque ligne, colonne et diagonale.

Lignes

- **Ligne 1** : `evaluate_line('X', 'O', 'X', 'X')`
 - Pas de trois 'X' alignés, pas de deux 'X' avec une cellule vide, pas de un 'X' avec deux cellules vides.
 - **Score** : 0
- **Ligne 2** : `evaluate_line(' ', 'X', 'O', 'X')`
 - Pas de trois 'X' alignés, pas de deux 'X' avec une cellule vide, pas de un 'X' avec deux cellules vides.
 - **Score** : 0
- **Ligne 3** : `evaluate_line(' ', ' ', 'X', 'X')`
 - Un 'X' avec deux cellules vides.
 - **Score** : 1

Colonnes

- **Colonne 1** : `evaluate_line('X', ' ', ' ', 'X')`
 - Un 'X' avec deux cellules vides.
 - **Score** : 1
- **Colonne 2** : `evaluate_line('O', 'X', ' ', 'X')`
 - Pas de trois 'X' alignés, pas de deux 'X' avec une cellule vide, pas de un 'X' avec deux cellules vides.
 - **Score** : 0
- **Colonne 3** : `evaluate_line('X', 'O', 'X', 'X')`
 - Pas de trois 'X' alignés, pas de deux 'X' avec une cellule vide, pas de un 'X' avec deux cellules vides.
 - **Score** : 0

Diagonales

- **Diagonale principale** : `evaluate_line('X', 'X', 'X', 'X')`
 - Trois 'X' alignés.
 - **Score** : 1000
- **Diagonale secondaire** : `evaluate_line('X', 'X', ' ', 'X')`
 - Deux 'X' avec une cellule vide.
 - **Score** : 10

Résumé des scores

$$\begin{aligned} \text{Lignes} &: 0 + 0 + 1 = 1 \\ \text{Colonnes} &: 1 + 0 + 0 = 1 \\ \text{Diagonales} &: 1000 + 10 = 1010 \end{aligned}$$

Score total Le score total est la somme de tous les scores des lignes, colonnes et diagonales :

$$1 + 1 + 1010 = 1012$$

Conclusion Lorsque vous exécutez `heuristique_alignement([['X', 'O', 'X'], [' ', 'X', 'O'], [' ', ' ', 'X']], 'X')`, le score renvoyé sera 1012. Cela reflète le fait qu'il y a une diagonale complète de 'X', ce qui est évalué très haut dans cette heuristique.

Voici une autre fonction d'heuristique qui attribue des valeurs absolues aux cases indépendamment du contexte. Cette approche consiste à attribuer des valeurs spécifiques aux cases du plateau en fonction de leur importance stratégique. Par exemple, les cases du centre et les coins peuvent avoir des valeurs plus élevées en raison de leur potentiel stratégique pour créer des alignements.

```

1 def heuristique_absolue(board, player):
2     valeurs = [
3         [3, 2, 3],
4         [2, 4, 2],
5         [3, 2, 3]
6     ]
7
8     score = 0
9     opposant = 'O' if player == 'X' else 'X'

```

```

10
11     for row in range(3):
12         for col in range(3):
13             if board[row][col] == player:
14                 score += valeurs[row][col]
15             elif board[row][col] == opponent:
16                 score -= valeurs[row][col]
17
18     return score

```

Solution

Solution 3.3.2

```

1 def meilleur_coup(board, player, heuristique):
2     best_score = -float('inf')
3     best_move = None
4
5     for row in range(3):
6         for col in range(3):
7             if board[row][col] == ' ':
8                 board[row][col] = player
9                 score = heuristique(board, player)
10                board[row][col] = ' '
11
12                if score > best_score:
13                    best_score = score
14                    best_move = (row, col)

```

Solution

Solution 3.3.2 Prenons [['X',' ','X'],[' ','O',' '],[' ','O',' ']]

```

1 >>> meilleur_coup([ ['X', ' ', 'X'],
2                    [' ', 'O', ' '],
3                    [' ', 'O', ' ']], 'X', heuristique_alignement)
4 (0, 1)
5
6 >>> meilleur_coup([ ['X', ' ', 'X'],
7                    [' ', 'O', ' '],
8                    [' ', 'O', ' ']], 'X', heuristique_absolue)
9 (2, 0)

```

C'est au joueur X de jouer; heuristique_alignement recommande la position gagnante : [['X', 'X', 'X'], [' ', 'O', ' '], [' ', 'O', ' ']]

Tandis que heuristique_absolue propose de jouer sur l'une des diagonales. Au coup suivant, 'O' peut gagner.

Solution

Solution 3.4

```

1 def jouer_tic_tac_toe(heuristique_X=heuristique_alignement,
2                      heuristique_O=heuristique_alignement, affiche=False):
3     board = [[' ' for _ in range(3)] for _ in range(3)]
4     current_player = 'X'
5     heuristiques = {'X': heuristique_X, 'O': heuristique_O}
6
7     for _ in range(9): # Il y a au plus 9 coups dans une partie de Tic Tac Toe
8         move = meilleur_coup(board, current_player, heuristiques[current_player])
9         if move:
10            board[move[0]][move[1]] = current_player
11            if affiche:
12                afficher_plateau(board)
13                print("\n")
14
15            if check_win(board, current_player):
16                if affiche:
17                    print(f"Le joueur {current_player} a gagné!")
18
19            return current_player

```

```
19         current_player = 'O' if current_player == 'X' else 'X'
20     else:
21         break # Si aucun coup n'est possible, terminer la boucle
22 if affiche:
23     print("La partie est un match nul!")
24 return None
25
```