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 allumette.
   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...

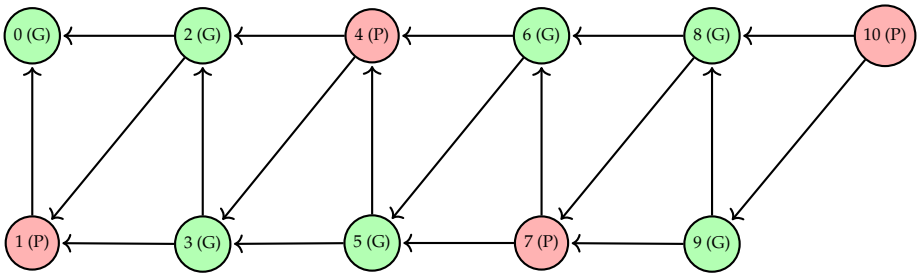
Remarque

On comprend à travers ces exemples que :

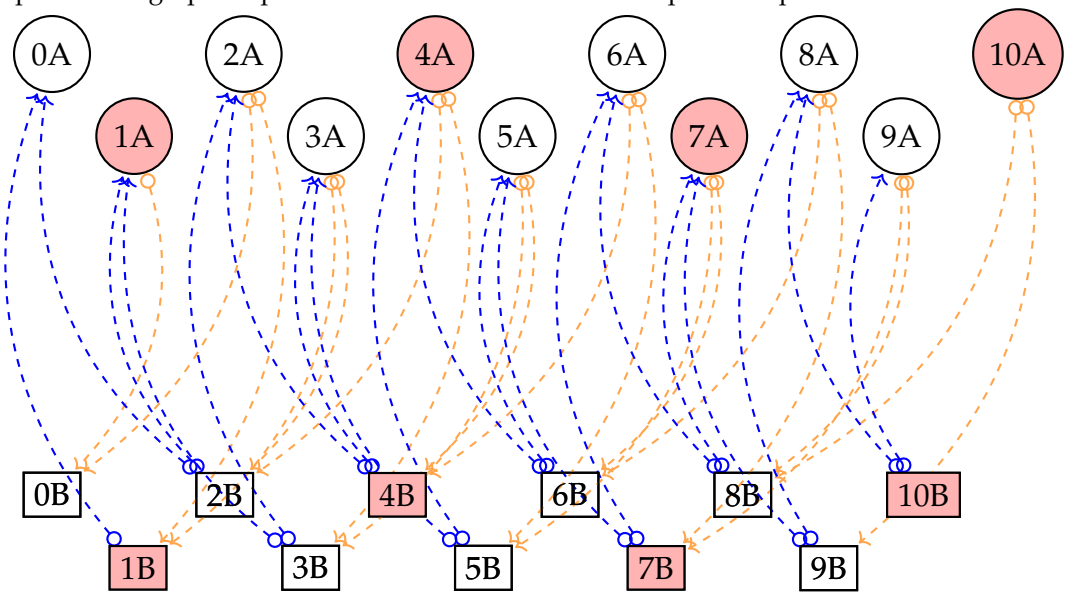
1. Certaines configurations sont définitivement perdantes, comme celle à 1 allumette.
2. Certaines configurations sont **gagnantes**, au sens où il **existe** un coup qui permet de laisser à l'autre joueur une position défavorable. Ici, 2 et 3 sont des position gagnantes : en ôtant respectivement 1 ou 2 allumettes, on laisse à ladversaire 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** mênent 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 partionné en deux sous–ensembles S_A et S_B tels que $S = S_A \cup S_B$ et $S_A \cap S_b = \varnothing$,
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.

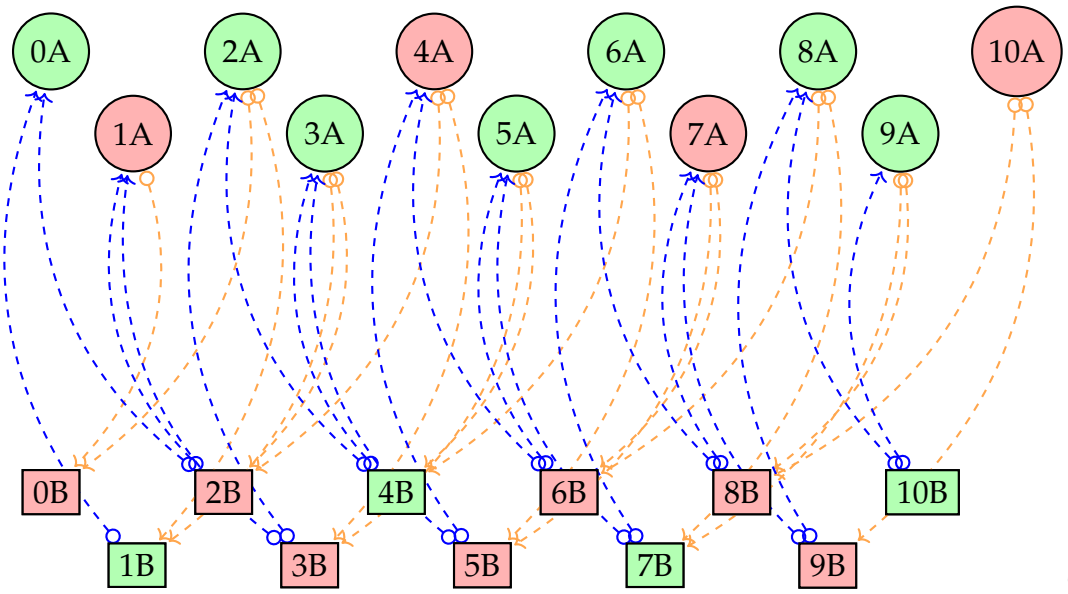
On peut par exemple penser au morpion, au puissance 4, aux dames, aux échecs, au go, etc.

▲ 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 $A_{n+1} := A_n \cup \{s \in S_A \mid \exists t \in A_n \; s \to t\} \cup \{s \in \S_b \mid \forall t \in S \; s \to t \Rightarrow t \in A_n\}$.
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)

l 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

Tour 1 (X)

X		

Tour 2 (O)

X		
	O	

Tour 3 (X)

X	X	
	O	

Tour 4 (O)

X	X	
	O	
O		

Tour 5 (X)

X	X	X
	O	
O		

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 chaines 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

La configuration :

O		X
	X	
O		

est représentée par la liste de listes de carcatères :

```
[['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 chaine de caractères 'X' ou 'O', et `board` est une position comme décrite précédemment.

```python
def check_win(board, player):
    # Verifier les lignes, colonnes et diagonales pour une victoire
    for row in range(3):
        if all([cell == player for cell in board[row]]):
            return True

    for col in range(3):
        if all([board[row][col] == player for row in range(3)]):
            return True

    if all([board[i][i] == player for i in range(3)]) or all([board[i][2 - i] == player
        for i in range(3)]):
        return True

    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.

```python

def evaluate_line(cell1, cell2, cell3, player):
    score = 0

    opponent = 'O' if player == 'X' else 'X'

    # Premier cas : toutes les cellules sont du joueur
    if cell1 == cell2 == cell3 == player:
        score = 1000
    # Deuxieme cas : deux cellules sont du joueur et une est vide
    elif cell1 == cell2 == player and cell3 == ' ':
        score = 10
    elif cell1 == cell3 == player and cell2 == ' ':
        score = 10
    elif A COMPLETER:
        score = 10
    # Troisieme cas : une cellule est du joueur et les autres sont vides
    elif cell1 == player and cell2 == cell3 == ' ':
        score = 1
    elif A COMPLETER:
        score = 1
    elif A COMPLETER:
        score = 1

    # Deduire les points si l'adversaire est en position similaire
    if A COMPLETER:
        score = -1000
    elif A COMPLETER:
        score = -10
    elif cell1 == cell3 == opponent and cell2 == ' ':
        score = -10
    elif cell2 == cell3 == opponent and cell1 == ' ':
        score =  A COMPLETER
    elif A COMPLETER:
        score = -1
    elif A COMPLETER:
        score = A COMPLETER
    elif A COMPLETER:
        score = A COMPLETER:
```

```
    return score
```

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

```
>>> board = [
    ['X', 'O', 'X'],
    [' ', 'X', 'O'],
    [' ', ' ', 'X']
]
>>> afficher_plateau(board)
 -------------
 | X | O | X |
 -------------
 |   | X | O |
 -------------
 |   |   | X |
 -------------
>>>  evaluate_line(board[2][0], board[2][1], board[2][2], 'X')
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	evaluate_line(board[0][0], board[0][1], board[0][2], 'X')
Ligne	(1, 0)-(1, 2)	, X, O	0	evaluate_line(board[1][0], board[1][1], board[1][2], 'X')
Ligne	(2, 0)-(2, 2)	, , X	1	evaluate_line(board[2][0], board[2][1], board[2][2], 'X')
Colonne	(0, 0)-(2, 0)	X, ,	1	evaluate_line(board[0][0], board[1][0], board[2][0], 'X')
Col	(0, 1)-(2, 1)	O, X,	0	evaluate_line(board[0][1], board[1][1], board[2][1], 'X')
Col	(0, 2)-(2, 2)	X, O, X	0	evaluate_line(board[0][2], board[1][2], board[2][2], 'X')
Diag	(0, 0)-(2, 2)	X, X, X	1000	evaluate_line(board[0][0], board[1][1], board[2][2], 'X')
Diag	(0, 2)-(2, 0)	X, X,	10	evaluate_line(board[0][2], board[1][1], board[2][0], 'X')

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

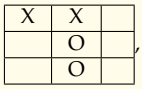
```
def heuristique_alignement(board, player):
    score = 0

    # Verifie les lignes
    for row in range(3):
        score += evaluate_line(board[row][0], board[row][1], board[row][2], player)

    # Verifie les colonnes
    for col in range(3):
        score += evaluate_line(board[0][col], board[1][col], board[2][col], player)

    # Verifie les diagonales
    score += evaluate_line(board[0][0], board[1][1], board[2][2], player)
    score += evaluate_line(board[0][2], board[1][1], board[2][0], player)

    return score
```

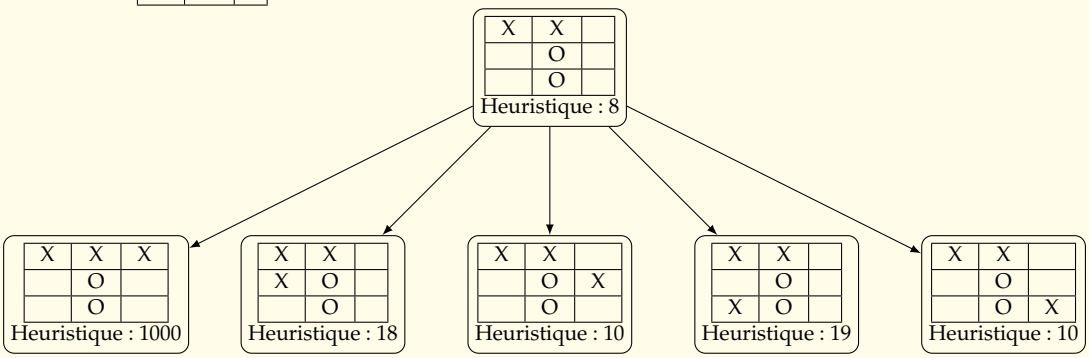
Que se passe-t-il si on éxécute `heuristique_alignement([ ['X', 'O', 'X'], [' ', 'X', 'O'], [' ', ' ', 'X'] ],'X')` ?
Proposer une autre fonction d'heuristique, appelée `heuristique_absolue` qui évalue la force d'une position en fonction des valeurs des cases occuppées indépendamment du contexte. On attribuera une valeur à chaque case en fonction de son avantage stratégique, et on sommera ou soustraira ces valeurs.

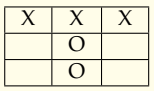
3.3.2 Choix machine du meilleur coup

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

Exemple

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



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

```
def jouer_tic_tac_toe_humain_vs_ordi():
    # Choisir qui commence
    first_player = input("Qui commence? (H pour humain, O pour ordinateur):
        ").strip().upper()
    while first_player not in ['H', 'O']:
        first_player = input("Entree invalide. Choisissez H pour humain ou O pour
            ordinateur: ").strip().upper()

    # Choisir l'heuristique
    print("Choisissez l'heuristique:")
    print("1. Heuristique alignement")
    print("2. Heuristique absolue")
    heuristique_choice = input("Entrez le numero de l'heuristique (1 ou 2): ").strip()
    while heuristique_choice not in ['1', '2']:
```

```
        heuristique_choice = input("Entree invalide. Entrez 1 ou 2: ").strip()

    heuristique = heuristique_alignement if heuristique_choice == '1' else
        heuristique_absolue

    board = [[' ' for _ in range(3)] for _ in range(3)]
    current_player = 'X' if first_player == 'H' else 'O'

    while True:
        afficher_plateau(board)

        if current_player == 'X':
            # Tour de l'humain
            valid_input = False
            while not valid_input:
                row = input("Entrez la ligne (0, 1, 2) : ")
                col = input("Entrez la colonne (0, 1, 2) : ")

                if row.isdigit() and col.isdigit():
                    row = int(row)
                    col = int(col)

                    if 0 <= row <= 2 and 0 <= col <= 2:
                        if board[row][col] == ' ':
                            board[row][col] = 'X'
                            valid_input = True
                        else:
                            print("Case deja occupee, essayez de nouveau.")
                    else:
                        print("Entree hors limites, veuillez entrer des nombres entre 0
                            et 2.")
                else:
                    print("Entree invalide, veuillez entrer des nombres entre 0 et 2.")
        else:
            # Tour de l'ordinateur
            move = meilleur_coup(board, 'O', heuristique)
            if move:
                board[move[0]][move[1]] = 'O'
            else:
                print("Aucun coup possible, la partie est terminee.")
                break

        if check_win(board, current_player):
            afficher_plateau(board)
            print(f"Le joueur {current_player} a gagne!")
            break

        if all(cell != ' ' for row in board for cell in row):
            afficher_plateau(board)
            print("La partie est un match nul!")
            break

        # Changer de joueur
        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

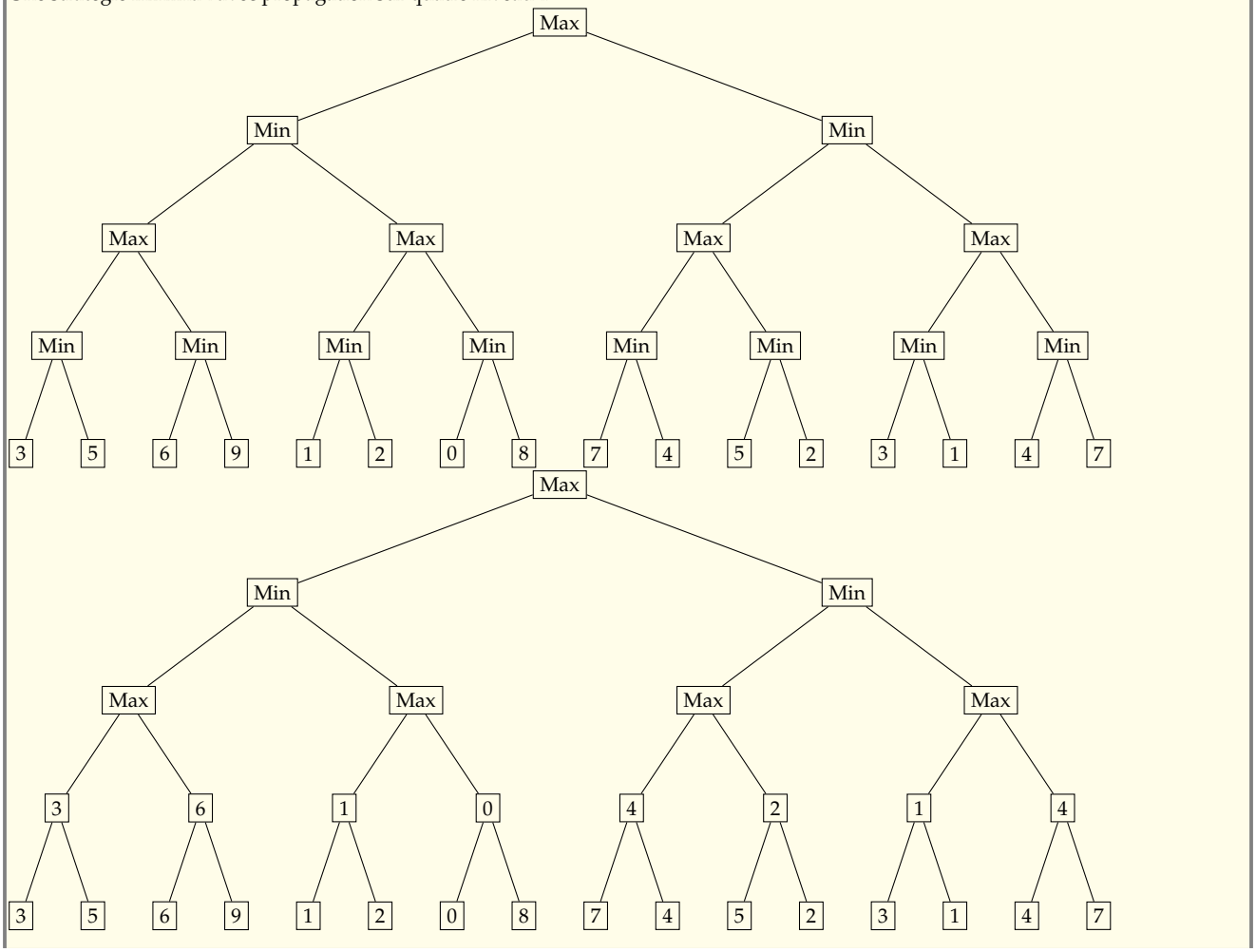
pour A). Ainsi, A peut anticiper les mouvements de B en prenant le maximum (niveau 1) des minima (niveau 2) des positions accessibles. Cette approche max-min permet de déterminer les mouvements optimaux en évaluant les résultats possibles de chaque action. Bien sûr, on peut utiliser cette méthode avec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

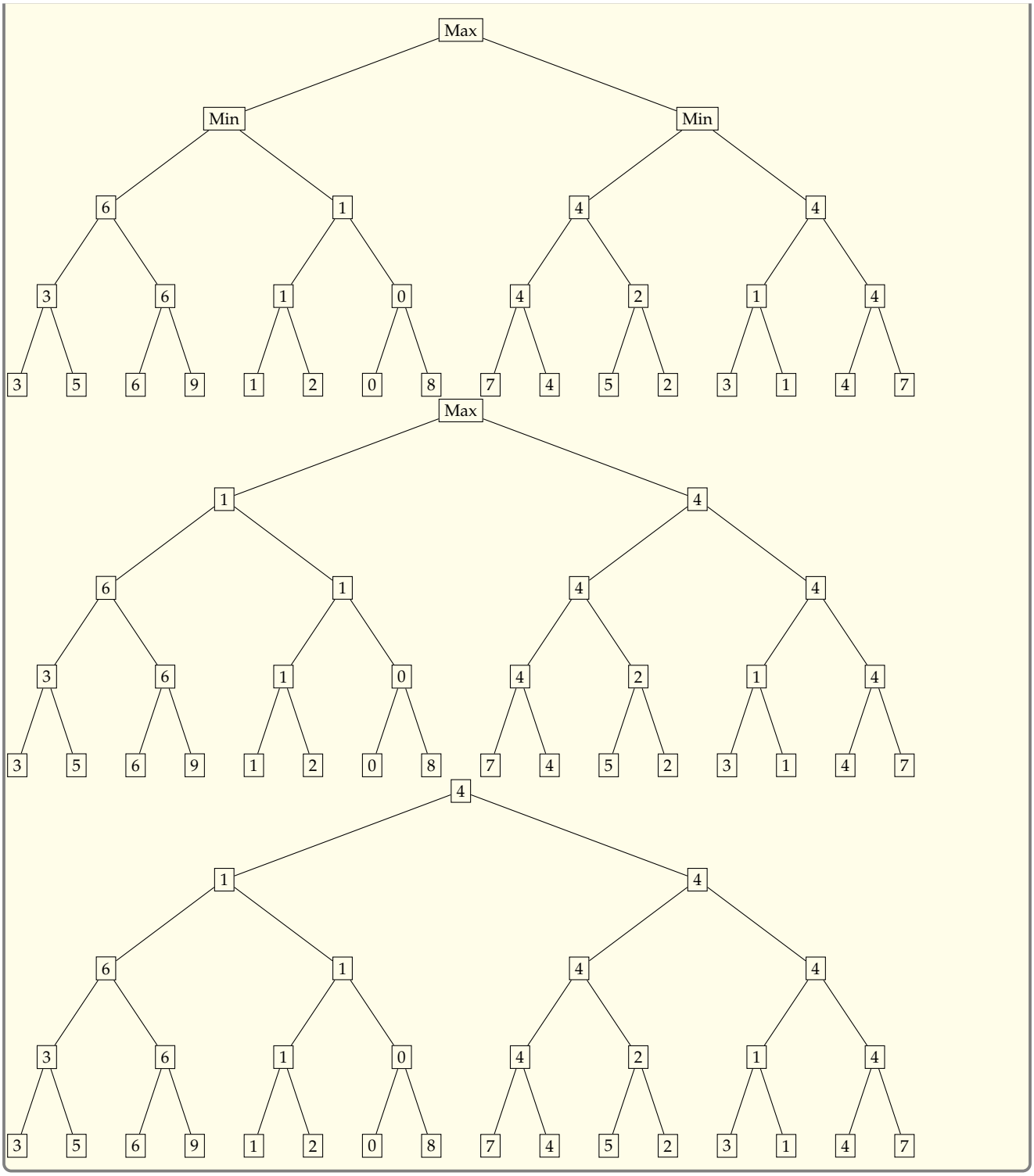
Le processus du Minimax se déroule comme suit :

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 par 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 aproche minimax ne fonctiionne 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.

L'algorithme `minimax` est une méthode couramment utilisée pour les jeux à deux joueurs, tels que le Puissance 4, afin de déterminer le meilleur coup possible. Dans cet article, nous expliquons une version simplifiée de l'algorithme `minimax`. Cette version explore toutes les branches de l'arbre de jeu jusqu'à une profondeur donnée.

Implémentons ce principe dans le cas du Puissance 4.

4.1 Puissance 4 : rappel des règles et mise en place

Le Puissance 4 est un jeu de stratégie pour deux joueurs.

Matériel de Jeu

- Un plateau de jeu de dimensions 7×6 (7 colonnes et 6 rangées).
- Deux types de jetons, généralement rouges X et jaunes O, un type par joueur.

Déroulement de la Partie

1. **Objectif** Le but du jeu est d'aligner quatre jetons de sa couleur, soit horizontalement, verticalement, ou en diagonale, avant l'adversaire.
2. **Début du jeu :** Chaque joueur choisit une couleur et décide qui commence.
3. **Tour de jeu :** Les joueurs jouent à tour de rôle.
   - À son tour, un joueur place un jeton dans une des colonnes du plateau.
   - Le jeton tombe dans la position la plus basse disponible de la colonne choisie.
4. **Objectif :** Le premier joueur qui aligne 4 jetons de sa couleur gagne.
5. **Égalité :** Si toutes les cases sont remplies sans qu'aucun joueur ne réalise un alignement de 4 jetons, la partie se termine par une égalité.

Conditions de Victoire

Un joueur gagne si, après avoir placé son jeton, il aligne (au moins) quatre jetons **consécutifs** de sa couleur dans l'une des directions suivantes :

- **Horizontalement :** Quatre jetons consécutifs sur une même ligne.
- **Verticalement :** Quatre jetons consécutifs dans une même colonne.
- **En diagonale :** Quatre jetons consécutifs alignés en diagonale (montante ou descendante).

Règles Supplémentaires

- Un coup est valide uniquement si la colonne choisie n'est pas pleine.
- Les joueurs doivent respecter l'ordre des tours et ne pas placer plusieurs jetons à la fois.

4.2 Implémentation

Exercice 9

1. Choix d'une représentation Python
2. Fonction d'affichage de plateau
3. Fonction d'Arbitrage
4. Proposition de plusieurs heuristiques
5. Implémentation de la fonction de choix d'un coup à partir d'une heuristique donnée.
6. Fonction qui organise un match IA/IA, ou bien IA/Joueur

4.3 Objectif de l'Algorithme

L'objectif principal de l'algorithme `minimax` est de :

- Maximiser le score pour le joueur actif (**Maximizer**).
- Minimiser le score pour l'adversaire (**Minimizer**).

4.4 Structure de la Fonction

La fonction `minimax` prend les paramètres suivants :

- `board` : L'état actuel du plateau (une matrice 2D).
- `depth` : La profondeur maximale de recherche.
- `maximizingPlayer` : Booléen indiquant si c'est au joueur Maximizer (`True`) ou Minimizer (`False`) de jouer.
- `player` : Le joueur actuel (par exemple, 'X').
- `heuristique_fn` : La fonction d'évaluation utilisée pour attribuer un score aux positions.

4.5 Code de l'Algorithme

Voici le code de l'algorithme en Python :

```python
def minimax(board, depth, maximizingPlayer, player, heuristique_fn):
    opponent = 'O' if player == 'X' else 'X'

    if depth == 0 or check_win(board, player) or check_win(board, opponent) or not any(' ' in
        row for row in board):
        return heuristique_fn(board, player)

    if maximizingPlayer:
        maxEval = -float('inf')
        for col in range(7):
            row = trouver_ligne_libre(board, col)
            if row is not None:
                board[row][col] = player
                eval = minimax_sans_elagage(board, depth - 1, False, player, heuristique_fn)
                board[row][col] = ' '
                maxEval = max(maxEval, eval)
        return maxEval
    else:
        minEval = float('inf')
        for col in range(7):
            row = trouver_ligne_libre(board, col)
            if row is not None:
                board[row][col] = opponent
                eval = minimax_sans_elagage(board, depth - 1, True, player, heuristique_fn)
                board[row][col] = ' '
                minEval = min(minEval, eval)
        return minEval
```

Listing 1 – Code Python de `minimax`

4.6 Explication de la Fonction

4.6.1 Condition d'Arrêt

La fonction s'arrête dans les cas suivants :

- La profondeur maximale de recherche (`depth`) est atteinte.
- Une condition de fin de partie est rencontrée (victoire ou égalité).
- Si l'une de ces conditions est remplie, la fonction retourne une évaluation numérique du plateau à l'aide de `heuristique_fn`.

4.6.2 Maximizing Player (Joueur Actif)

Lorsque c'est au tour du joueur actif (Maximizer), l'objectif est de maximiser le score évalué du plateau. La logique est la suivante :

1. **Initialisation :** Le score maximum potentiel (`maxEval`) est initialisé à une valeur très faible ($-\infty$).
2. **Parcours des colonnes :** On parcourt toutes les colonnes possibles du plateau (de 0 à 6) :
   - Trouver la première ligne libre de la colonne (avec `trouver_ligne_libre`).
   - Si un coup est possible, simuler ce coup en plaçant un jeton pour le joueur actif.
3. **Appel récursif :** Appeler `minimax` pour évaluer les réponses potentielles de l'adversaire (Minimizer) à une profondeur réduite.
4. **Annulation du coup :** Après l'évaluation, remettre le plateau dans son état initial pour tester les autres colonnes.
5. **Mise à jour du score :** Comparer la valeur retournée par l'appel récursif avec `maxEval`, et conserver le plus grand score.

À la fin de la boucle, `maxEval` contient le meilleur score que le Maximizer peut obtenir. La fonction retourne cette valeur.

4.6.3 Minimizing Player (comportement simulé par Joueur actif

Le Minimizer cherche à minimiser le score ; de même :

1. Initialiser `minEval` à $+\infty$.
2. Pour chaque colonne possible, simuler un coup.
3. Appeler récursivement `minimax` pour le Maximizer.
4. Mettre à jour `minEval` avec le pire score trouvé.
5. Retourner `minEval`.

5 Solutions

Solution

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.

```
def heuristique_absolue(board, player):
    valeurs = [
        [3, 2, 3],
        [2, 4, 2],
        [3, 2, 3]
    ]

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

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

Solution

Solution 4.2

```
1  def afficher_plateau(board,avec_pause=False):
2      for row in board:
3          print(' | '.join(row))
4          print('-' * 29)
5      if avec_pause:
6          input('press to continue')
7
8  def check_win(board, player):
9      # Verifier les lignes horizontales
10     for row in range(6):
11         for col in range(4):
12             if all(board[row][col + i] == player for i in range(4)):
13                 return True
14
15     # Verifier les lignes verticales
16     for row in range(3):
17         for col in range(7):
18             if all(board[row + i][col] == player for i in range(4)):
19                 return True
20
21     # Verifier les diagonales montantes
22     for row in range(3):
23         for col in range(4):
24             if all(board[row + i][col + i] == player for i in range(4)):
25                 return True
26
27     # Verifier les diagonales descendantes
28     for row in range(3, 6):
29         for col in range(4):
30             if all(board[row - i][col + i] == player for i in range(4)):
31                 return True
32
33     return False
34
35 def trouver_ligne_libre(board, col):
36     for row in range(5, -1, -1):
37         if board[row][col] == ' ':
38             return row
39     return None
40
41 def evaluate_segment(board, start_row, start_col, delta_row, delta_col, player):
42     score = 0
43     opponent = 'O' if player == 'X' else 'X'
44     player_count = 0
45     opponent_count = 0
46     empty_count = 0
47
48     for i in range(4):
49         cell = board[start_row + i * delta_row][start_col + i * delta_col]
50         if cell == player:
51             player_count += 1
52         elif cell == opponent:
53             opponent_count += 1
54         else:
55             empty_count += 1
56
57     if player_count == 4:
58         score += 100000
59     elif player_count == 3 and empty_count == 1:
60         score += 100
61     elif player_count == 2 and empty_count == 2:
62         score += 10
63     elif player_count == 1 and empty_count == 3:
64         score += 1
```

```
65
66      if opponent_count == 4:
67          score -= 100000
68      elif opponent_count == 3 and empty_count == 1:
69          score -= 100
70      elif opponent_count == 2 and empty_count == 2:
71          score -= 10
72      elif opponent_count == 1 and empty_count == 3:
73          score -= 1
74
75      return score
76
77
78  def heuristique(board, player):
79      # Fonction heuristique initiale
80      score = 0
81
82      # Verifier les lignes horizontales
83      for row in range(6):
84          for col in range(4):
85              score += evaluate_segment(board, row, col, 0, 1, player)
86
87      # Verifier les lignes verticales
88      for row in range(3):
89          for col in range(7):
90              score += evaluate_segment(board, row, col, 1, 0, player)
91
92      # Verifier les diagonales montantes
93      for row in range(3):
94          for col in range(4):
95              score += evaluate_segment(board, row, col, 1, 1, player)
96
97      # Verifier les diagonales descendantes
98      for row in range(3, 6):
99          for col in range(4):
100             score += evaluate_segment(board, row, col, -1, 1, player)
101
102     return score
103
104 def heuristique2(board, player):
105     # Fonction heuristique 2 qui privilegie les cases centrales
106     position_scores = [
107         [3, 4, 5, 7, 5, 4, 3],
108         [4, 6, 8, 10, 8, 6, 4],
109         [5, 8, 11, 13, 11, 8, 5],
110         [5, 8, 11, 13, 11, 8, 5],
111         [4, 6, 8, 10, 8, 6, 4],
112         [3, 4, 5, 7, 5, 4, 3]
113     ]
114
115     score = 0
116     for row in range(6):
117         for col in range(7):
118             if board[row][col] == player:
119                 score += position_scores[row][col]
120             elif board[row][col] != ' ':
121                 score -= position_scores[row][col]
122
123     return score
124
125 def heuristique3(board, player):
126     position_scores = [
127         [3, 4, 5, 7, 5, 4, 3],
128         [4, 6, 8, 10, 8, 6, 4],
129         [5, 8, 11, 13, 11, 8, 5],
130         [5, 8, 11, 13, 11, 8, 5],
131         [4, 6, 8, 10, 8, 6, 4],
132         [3, 4, 5, 7, 5, 4, 3]
133     ]
134
135     segment_score = heuristique(board, player)
136     position_score = 0
137
138     for row in range(6):
139         for col in range(7):
140             if board[row][col] == player:
141                 position_score += position_scores[row][col]
```

```
142                 elif board[row][col] != ' ':
143                     position_score -= position_scores[row][col]
144 
145     return segment_score + position_score
```

Listing 2 – Code Python de mise en place du Puissance 4