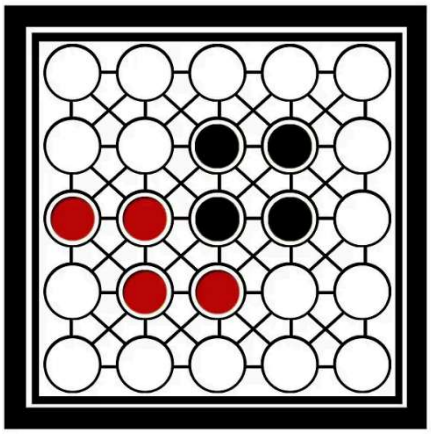


RAPPORT DE PROJET DU JEU TEEKO

Ivann Vasic

Lucas Vigny

Emile Lagarde

Automne 2025

Table des matières

1/Introduction

Teeko est un jeu de société stratégique pour deux joueurs qui se déroule sur un plateau de 5x5. Dans le domaine de l'intelligence artificielle, les jeux de société ont été utilisés comme terrain d'essai pour la modification et l'évaluation d'algorithmes depuis longtemps. Ils permettent de tester des idées liées à la prise de décision, à la recherche heuristique et à l'optimisation

Objectif du projet

Le principal objectif de notre projet est de mettre en œuvre une simulation complète du jeu Teeko en langage de programmation Python. Cette simulation contient également une interface graphique simple et conviviale afin de rendre le jeu plus participatif, ainsi qu'un module d'intelligence artificielle qui peut rivaliser contre un joueur humain. L'IA doit être évolutive, c'est-à-dire conçue avec plusieurs niveaux de difficulté : facile, intermédiaire et difficile.

Pour chacun des niveaux de difficulté choisis, le niveau intelligent correspondant change également, allant de mouvements aléatoires à des algorithmes plus complexes tels que l'algorithme Minimax avec élagage alpha-bêta. Cela permet d'évaluer les caractéristiques de l'IA ainsi que sa capacité à faire face et à fonctionner par rapport aux différents styles de jeu humains.

Présentation des défis liés à la programmation

Voici les principaux défis techniques et conceptuels que nous avons rencontré lors de la réalisation de notre projet.

- **Programmation de l'IA** : Développer un bot capable de jouer efficacement à Teeko a été un défi majeur. Cela a nécessité une bonne compréhension et l'application d'algorithmes de recherche adaptés. Trouver un équilibre entre efficacité et rapidité a été particulièrement délicat.
- **Interface utilisateur** : La conception d'une interface graphique intuitive et réactive avec Tkinter a demandé du temps et de la réflexion. Il était important qu'elle gère de manière fluide les interactions des joueurs, les phases de placement et de déplacement, tout en reflétant l'état du plateau de manière claire.
- **Gestion des règles et des états du jeu** : L'un des aspects les plus complexes du projet a été d'implémenter les différentes règles du jeu. Cela comprenait la validation des mouvements, la détection des configurations gagnantes, la transition entre les phases du jeu, et la gestion des éventuelles entrées incorrectes de l'utilisateur pour garantir la robustesse du programme.0

2/ Spécification et Formalisation du problème

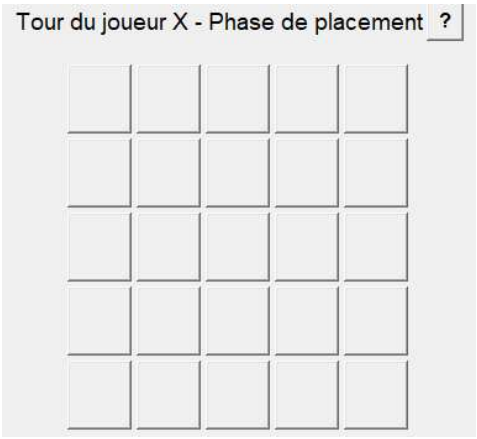
Représentation du plateau

Nous avons choisi de représenter le plateau de jeu sous la forme d'une matrice 5x5 en Python. Chaque case est initialement vide et représentée par un espace (‘ ‘). Voici un exemple de la structure utilisée :

1/ représentation de la structure du plateau

```
plateau = [
    [' ', ' ', ' ', ' ', ' '],
    [' ', ' ', ' ', ' ', ' '],
    [' ', ' ', ' ', ' ', ' '],
    [' ', ' ', ' ', ' ', ' '],
    [' ', ' ', ' ', ' ', ' ']
]
```

2/ représentation graphique du plateau



Cette structure permet d'accéder directement aux cases via leurs indices (ligne, colonne) et de mettre à jour les états en fonction des actions des joueurs.

Chaque joueur est représenté par un caractère unique : ‘X’ pour le premier joueur et ‘O’ pour le second. Ce format facilite la gestion des déplacements, la vérification des conditions de victoire, et la mise à jour du plateau après chaque tour.

Phases du jeu

- **Phase de placement :**
  Pendant cette phase, chaque joueur place à tour de rôle ses quatre pions sur des cases libres.
  Le plateau est mis à jour en fonction des coordonnées choisies, et les positions sont enregistrées dans la liste des pions du joueur concerné.
- **Phase de déplacement :**
  Une fois les pions placés, les joueurs déplacent à tour de rôle un de leurs pions vers une case adjacente libre.
  Les règles de déplacement autorisent des mouvements dans les huit directions (horizontal, vertical, diagonal).

Objectif de l'IA :

- **Gagner la partie** : L'IA identifie les configurations gagnantes possibles et cherche à les réaliser en minimisant les opportunités de l'adversaire.
- **Empêcher l'adversaire de gagner** : Grâce à l'utilisation de simulations, l'IA prévoit les coups adverses potentiels et bloque les situations menaçantes.

Voici comment l'IA as été programmé en fonctions des différents modes de jeu :

Facile : L'IA joue de manière aléatoire.

Intermédiaire : L'IA utilise l'algorithme Minimax avec une profondeur de 1, de sorte à qu'il puisse contrer les coups faciles ou gagner si le joueur n'est pas attentif.

Difficile : L'IA utilise l'algorithme Minimax avec une profondeur de 4 avec élagage alpha-bêta pour calculer les meilleurs coups possibles.

L'un des principaux défis était de trouver un compromis entre la performance de l'IA et les temps de calcul, notamment sur un plateau de 5x5 où les combinaisons de mouvements augmentent rapidement.

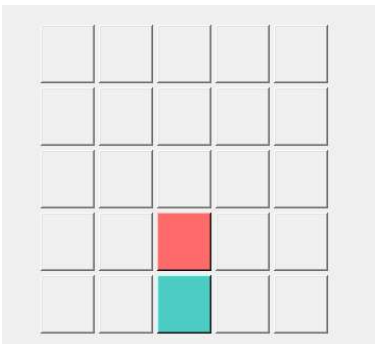
3/Logique principale du jeu

Déroulement du jeu

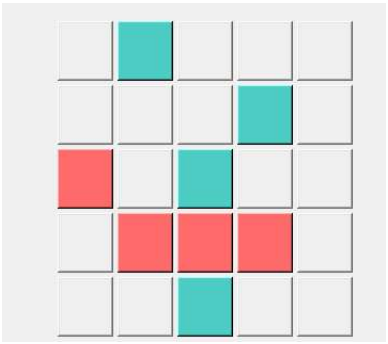
- Initialisation :

Au début, le plateau est vide et les listes des positions des pions des joueurs sont vides.

- Placement des pions :

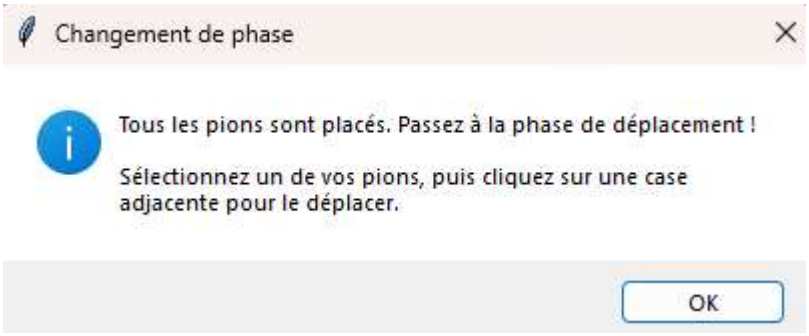


Lorsqu'un joueur clique sur une case vide, la position est ajoutée à la liste des pions du joueur concerné, et le plateau est mis à jour.

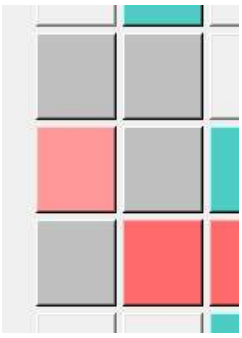


- Fin de phase de placement :

La phase de placement se termine lorsque les deux joueurs ont placé leurs quatre pions. On en informe ainsi le joueur.



- Sélection du pion :

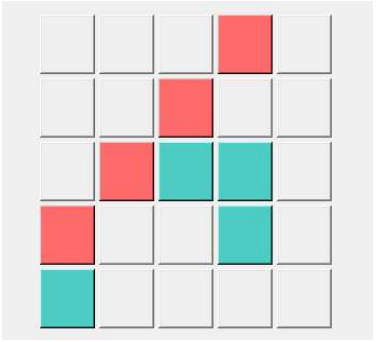


Une fois les 4 premiers pions placés, les joueurs déplacent à tour de rôle leurs pions sur des cases adjacentes libres. Le joueur sélectionne un pion en cliquant dessus. La case correspondante est mise en surbrillance, et les cases disponibles sont grisées pour indiquer les emplacements possibles pour la pièce.

- Déplacement du pion :

Une fois la case de destination choisie, le pion est déplacé et le plateau mis à jour. Puis le déplacement est validé seulement si la case de destination est vide et adjacente.

- Fin de partie :



La partie se termine lorsque l'algorithme détecte qu'un joueur gagne. Il en est alors averti et la partie se termine.

Vérification des Conditions de Victoire

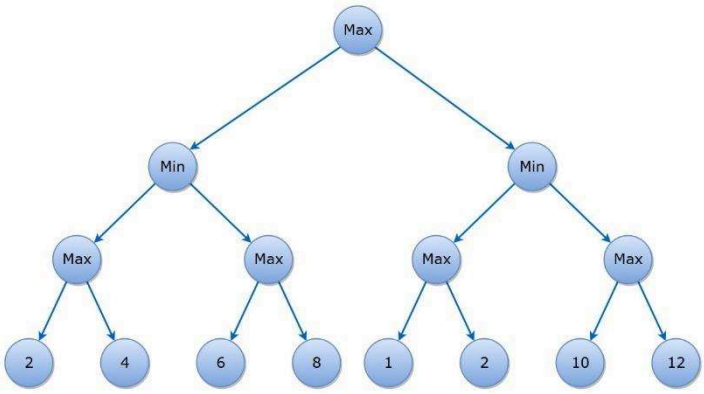
Pour garantir une détection correcte des configurations gagnantes, nous avons divisé ce problème en quatre sous-parties correspondant aux types de victoires possibles :

- **Alignements horizontaux** : Pour chaque ligne du plateau, nous avons vérifié si quatre cases consécutives contenaient le même symbole (‘X’ ou ‘O’).
- **Alignements verticaux** : Nous avons appliqué une logique similaire aux colonnes.
- **Alignements diagonaux** : Deux cas ont été considérés, la diagonale descendante et la diagonale montante.
- **Carrées** : Nous avons vérifié si les quatre cases d’un sous-ensemble 2x2 étaient identiques.

Niveaux de Difficulté de l’IA

Pour le niveau **moyen**, la logique principale repose sur le fait que l’IA tente de bloquer les coups gagnants adverses et privilégie les positions stratégiques comme le centre et les cases adjacentes en utilisant le même algorithme que pour le mode difficile mais avec une profondeur de 1.

Pour le niveau **difficile**, L’IA utilise l’algorithme Minimax avec un élagage alpha-bêta pour évaluer les meilleurs coups. Cet algorithme est très régulièrement utilisé en intelligence artificielle. L’algorithme Minimax alterne entre Maximisation (Cherche à maximiser les gains pour l’IA) et la Minimisation (Cherche à minimiser les gains de l'adversaire). Tandis que l’élagage alpha-bêta permet d' optimiser Minimax en évitant d'explorer des branches inutiles.



Gestion des Interactions Utilisateur

- **Capture des clics** : Chaque bouton du plateau est lié à une fonction qui traite les coordonnées correspondantes.
- **Mise à jour de l’interface** : Après chaque action, l’interface graphique est mise à jour pour refléter l’état actuel du jeu.

- **Validation des actions** : Les clics invalides (par exemple, cliquer sur une case occupée ou hors du plateau) sont ignorés avec un message d'erreur.

Nous avons utilisé le module Tkinter pour concevoir l'interface graphique du jeu Teeko.

Les cases libres sont affichées avec une couleur neutre ('SystemButtonFace'), tandis que les cases occupées changent de couleur pour indiquer le joueur correspondant : rouge pour le joueur 1 et bleu pour joueur 2.

Une barre de statut en haut de l'interface informe les joueurs de l'état actuel du jeu (par exemple : "Tour du joueur X - Phase de placement"). Cette fonctionnalité va se mettre à jour dynamiquement après chaque action.

Nous avons également mis en place un menu d'aide qui affiche les règles du jeu au lancement du programme pour que les nouveaux joueurs puissent connaître les règles.

Nous avons opté pour une interface simple et intuitive afin que les joueurs puissent comprendre rapidement comment interagir avec le jeu.

Pour éviter les erreurs (comme cliquer sur une case déjà occupée), nous avons ajouté des messages d'erreur via des boîtes de dialogue.

4/ Difficultés rencontrées

- Nous avons rencontré des problèmes de performance lorsque l'IA évaluait toutes les possibilités sur un plateau de 5x5. La complexité augmentant rapidement, il a fallu limiter la profondeur de recherche à 4 pour maintenir des temps de calcul permettant le déroulement fluide de la partie. L'implémentation de l'élagage alpha-bêta a permis cependant de réduire considérablement ces problèmes.
- Nous avons eu du mal à trouver comment bien jauger l'état de la partie avec evaluate_board. Il faut trouver un moyen de prendre en compte beaucoup de possibilités, même si elles ne sont pas gagnantes et attribuer le bon score à chaque possibilité en fonction de si le joueur peut gagner ou non.
- Nous avons rencontré des problèmes lorsque les utilisateurs effectuaient des actions invalides, comme cliquer sur une case déjà occupée ou hors du plateau. Il a fallu ajouter des messages d'erreur explicites et bloquer ces interactions sans interrompre l'expérience de jeu.
- Nous avons également rencontré des lacunes lors de l'apprentissage de l'utilisation de Tkinter. Étant notre première expérience avec les interfaces en langage python nous avons dû apprendre de nombreuses notions que l'on ne maîtrisait pas.

5/ Améliorations possibles et perspectives

Optimisation des Performances de l'IA

Optimisation de la fonction Evaluate_Board qui manque parfois des situations plutôt faciles à voir mais que nous n'avons pas réussi à coder.

Utilisation de techniques avancées comme les tables de transposition pour mémoriser les états déjà explorés et réduire le nombre de calculs.

Une amélioration encore plus poussée et sûrement très intéressante mais beaucoup plus complexe serait d'incorporer des algorithmes d'apprentissage par renforcement (comme Q-learning) pour permettre à l'IA d'apprendre des parties jouées.

Ajout de Fonctionnalités

- **Modes de jeu supplémentaires :**

Réseau local : Permettre à deux joueurs de jouer sur différents ordinateurs connectés sur un réseau local.

- **Parties enregistrées :**

Intégrer une fonctionnalité pour sauvegarder les parties en cours et les rejouer plus tard, avec par exemple un format de sauvegarde basé sur JSON pour stocker l'état du plateau et des joueurs.

- **Statistiques et analyses post-partie** :

Affichages de différentes statistiques à la fin de la partie : (nombre de coup joué par les joueurs, taux de contrôles des zones stratégiques …)

- **Amélioration du Visuel de l'Interface**

Thèmes personnalisables :

Ajouter des options de personnalisation de l'interface, comme des palettes de couleurs et des icônes pour remplacer les pions ('X' et 'O').

Correction de Certains Bugs

- Synchronisation des clics et de l’affichage :

Parfois, les clics rapides provoquent des désynchronisations entre l’état logique et l’état visuel. Une file d’attente pour traiter les événements utilisateur pourrait résoudre ce problème.

- Améliorations du mode difficile :

Le mode difficile engendre encore certains bugs quelques fois. Il est possible d’améliorer encore de nombreux points à ce sujet.

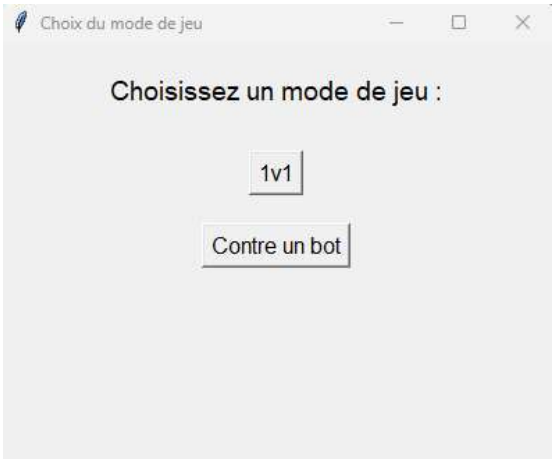
Perspectives d’ouverture du projet

Nous pouvons envisager d'améliorer le jeu en intégrant une IA apprenante utilisant l'apprentissage par renforcement pour ajuster ses stratégies en fonction des parties jouées, tout en rendant l'IA dynamique et adaptable au style de jeu de l'adversaire (agressif, défensif, ou aléatoire). Nous pouvons également considérer optimiser les performances en adoptant des algorithmes hybrides combinant Minimax et des approches probabilistes. Par ailleurs, l'expérience multijoueur serait enrichie par l'ajout d'un mode en ligne avec classements, tournois, et serveurs centralisés ou p2p. Enfin, le développement d'un mode campagne scénarisé, proposant des défis progressifs, et l'organisation de tournois locaux ou en ligne, permettraient de stimuler la communauté des joueurs et d'élargir l'attrait du jeu.

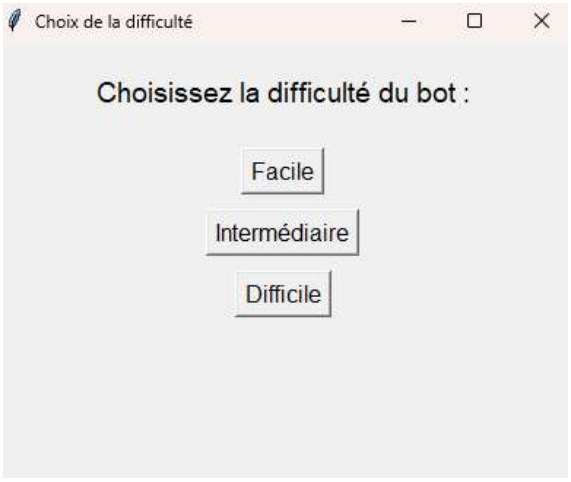
Pour optimiser la vitesse de recherche de l’algorithme, nous aurions pu aussi utiliser des threads qui travaillent ensemble afin de trouver la solution beaucoup plus rapidement.

Annexe 1 : Description d’une situation détaillée

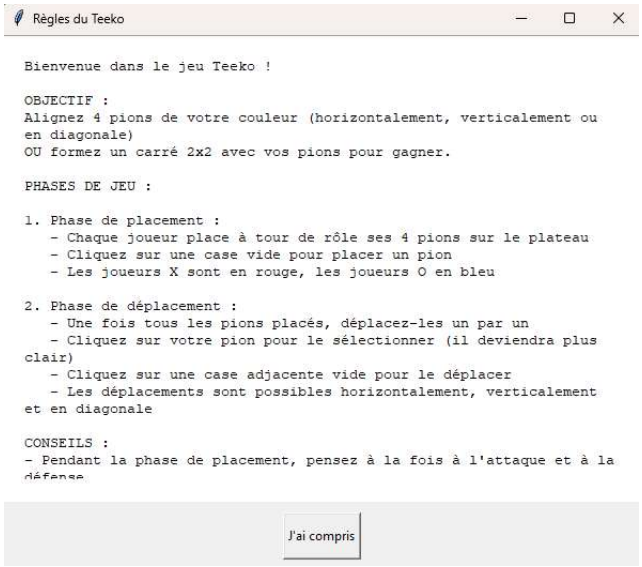
Dans cette partie, nous allons réaliser un exemple d’une situation réel d’un joueur jouant à notre jeu Teeko.



Le joueur commence par choisir le mode de jeu et décide de jouer contre un bot.

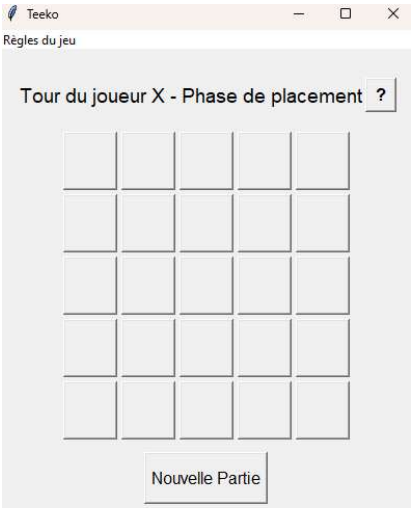


Il choisit ensuite la difficulté et décide de prendre l'intermédiaire.

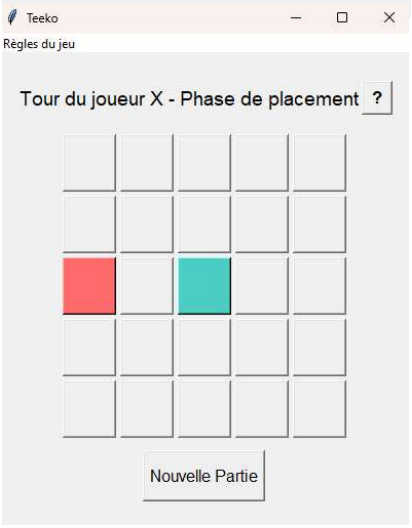


Puis les règles du jeu vont s’afficher.

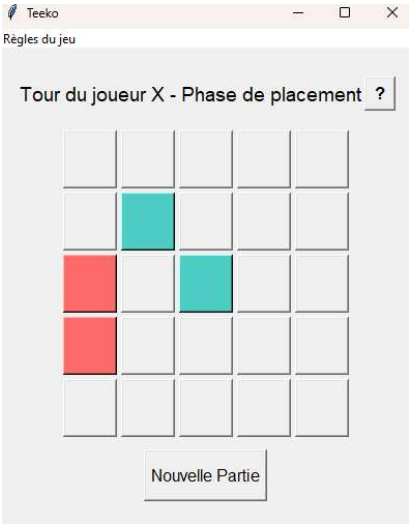
Le joueur clique sur j’ai compris.



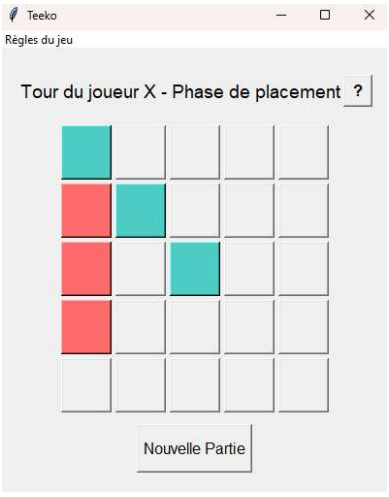
Voici alors l'interface de la partie.



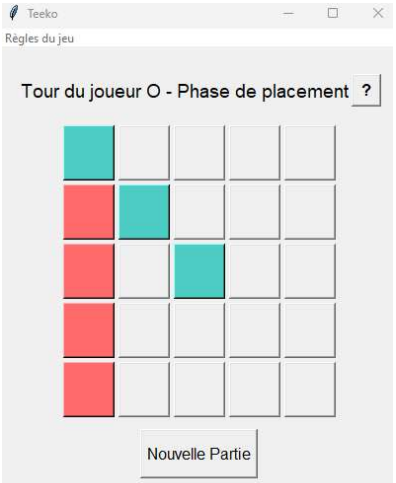
Le joueur positionne le premier pion de couleur rouge et l'ia répond en bloquant la ligne.



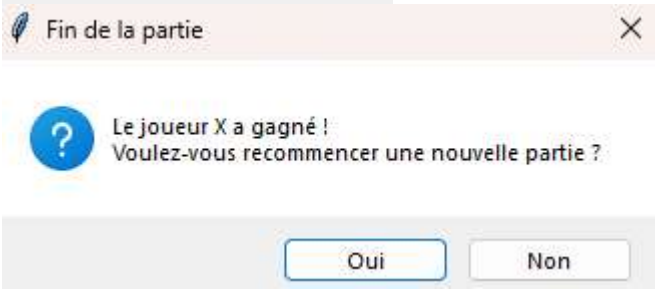
Le joueur positionne ainsi le deuxième pion, en tentant de faire une colonne. L'ia répond également.



Le joueur place maintenant le troisième pion, et l'ia s'empresse de le bloquer.



Mais au prochain tour, le joueur réussi à former une colonne et remporte la partie.



Ainsi se message s'affiche.

Si le joueur appuie sur oui, l'interface se réinitialise et s'il appuie sur non, le jeu se ferme.

Annexe 2 : Description détaillé de l'algorithme Minimax

Dans cette annexe nous fournissons une explication détaillée du fonctionnement de l'algorithme Minimax.

Fonction evaluate_board

```
def evaluate_board(self):
        score = 0
        score += self.evaluate_alignments(self.player)
        score -= self.evaluate_alignments(self.opponent)
        for x in range(5):
            for y in range(5):
                if self.board[x][y] == self.player and 1 <= x <= 3 and 1 <= y <= 3:
                    score += 2
        return score
```

Elle fournit une évaluation heuristique de l'état actuel du plateau. Elle attribue des scores en fonction des critères suivants :

- Alignements du joueur :

Attribue des points pour des alignements de 2, 3, ou 4 pions, les alignements complets ayant un score élevé

- Blocage des alignements adverses :

Diminue le score si l'adversaire est proche d'un alignement gagnant

- Contrôle des zones stratégiques :

Ajoute des points pour les pions placés au centre ou dans des positions avantageuses.

Fonction minimax

```
def minimax(self, depth, is_maximizing, alpha, beta):
        winner = self.check_win()
        if winner:
            return 1000 if winner == self.player else -1000  #
Augmentation de la valeur de victoire
        if depth == 0:
            return self.evaluate_board()

        if is_maximizing:
            max_eval = float('-inf')
            # Simuler les placements ET les mouvements
            if len(self.pieces[self.player]) < 4:  # Phase de placement
                moves = [(i, j) for i in range(5) for j in range(5) if
self.board[i][j] == ' ']
                for x, y in moves:
                    self.board[x][y] = self.player
                    self.pieces[self.player].append((x, y))
                    eval = self.minimax(depth - 1, False, alpha, beta)
                    self.board[x][y] = ' '
                    self.pieces[self.player].remove((x, y))
                    max_eval = max(max_eval, eval)
                    alpha = max(alpha, eval)
                    if beta <= alpha:
                        break
            else:  # Phase de mouvement
                for x, y in self.pieces[self.player]:
                    for i in range(max(0, x - 1), min(5, x + 2)):
                        for j in range(max(0, y - 1), min(5, y + 2)):
                            if self.board[i][j] == ' ':
                                self.board[x][y], self.board[i][j] = ' ',
self.player
                                self.pieces[self.player].remove((x, y))
                                self.pieces[self.player].append((i, j))
                                eval = self.minimax(depth - 1, False,
alpha, beta)
                                self.board[x][y], self.board[i][j] =
self.player, ' '
                                self.pieces[self.player].remove((i, j))
                                self.pieces[self.player].append((x, y))
                                max_eval = max(max_eval, eval)
                                alpha = max(alpha, eval)
                                if beta <= alpha:
                                    break
            return max_eval
        else:
            min_eval = float('inf')
            # Même chose pour l'adversaire
            if len(self.pieces[self.opponent]) < 4:
                moves = [(i, j) for i in range(5) for j in range(5) if
self.board[i][j] == ' ']
                for x, y in moves:
                    self.board[x][y] = self.opponent
                    self.pieces[self.opponent].append((x, y))
                    eval = self.minimax(depth - 1, True, alpha, beta)
                    self.board[x][y] = ' '
                    self.pieces[self.opponent].remove((x, y))
                    min_eval = min(min_eval, eval)
                    beta = min(beta, eval)
                    if beta <= alpha:
```

```
                        break
            else:
                for x, y in self.pieces[self.opponent]:
                    for i in range(max(0, x - 1), min(5, x + 2)):
                        for j in range(max(0, y - 1), min(5, y + 2)):
                            if self.board[i][j] == ' ':
                                self.board[x][y], self.board[i][j] = ' ',
self.opponent
                                self.pieces[self.opponent].remove((x, y))
                                self.pieces[self.opponent].append((i, j))
                                eval = self.minimax(depth - 1, True,
alpha, beta)
                                self.board[x][y], self.board[i][j] =
self.opponent, ' '
                                self.pieces[self.opponent].remove((i, j))
                                self.pieces[self.opponent].append((x, y))
                                min_eval = min(min_eval, eval)
                                beta = min(beta, eval)
                                if beta <= alpha:
                                    break
            return min_eval
```

L'algorithme Minimax explore toutes les possibilités pour maximiser les gains de l'IA tout en minimisant ceux de l'adversaire. Il utilise l'élagage alpha-bêta pour améliorer les performances.

Conditions de terminaison : Si un gagnant est trouvé, la fonction retourne un score élevé (+100 pour l'IA, -100 pour l'adversaire).

Si la profondeur atteinte est 0, la fonction retourne une évaluation heuristique de l'état actuel.

Maximisation (IA) : Parcourt toutes les cases libres du plateau.

Simule un mouvement de l'IA et appelle minimax pour les tours suivants.

Met à jour le meilleur score (max_eval) et la valeur alpha.

Arrête l'exploration des branches inutiles si alpha ≥ beta.

Minimisation (Adversaire) :

Similaire à la maximisation, mais en simulant les mouvements de l'adversaire.

Met à jour le pire score possible (min_eval) et la valeur bêta.

Fonction movement_hard et best_move

```
 def movement_hard(self):
        return self.best_move(depth=4, move_type="movement")

def best_move(self, depth, move_type="placement"):
        best_score = float('-inf')
        best_action = None

        # Generate possible actions
        actions = (
            [(i, j) for i in range(5) for j in range(5) if self.board[i][j] == ' ']
            if move_type == "placement" else
            [
                (x, y, i, j)
                for x, y in self.pieces[self.player]
                for i in range(max(0, x - 1), min(5, x + 2))
                for j in range(max(0, y - 1), min(5, y + 2))
                if self.board[i][j] == ' '
            ]
        )

        # Check for immediate winning move
        for action in actions:
            if move_type == "placement":
                x, y = action
                self.board[x][y] = self.player
            else:
                x, y, i, j = action
                self.board[x][y], self.board[i][j] = ' ', self.player
                self.pieces[self.player].remove((x, y))
                self.pieces[self.player].append((i, j))

            if self.check_win() == self.player:
                # Undo changes and immediately return the winning move
                if move_type == "placement":
                    self.board[x][y] = ' '
                else:
                    self.board[x][y], self.board[i][j] = self.player, ' '
                    self.pieces[self.player].remove((i, j))
                    self.pieces[self.player].append((x, y))
                return action

            # Undo changes
            if move_type == "placement":
                self.board[x][y] = ' '
```

```
            else:
                self.board[x][y], self.board[i][j] = self.player, ' '
                self.pieces[self.player].remove((i, j))
                self.pieces[self.player].append((x, y))

        # Explore moves using minimax
        for action in actions:
            if move_type == "placement":
                x, y = action
                self.board[x][y] = self.player
            else:
                x, y, i, j = action
                self.board[x][y], self.board[i][j] = ' ', self.player
                self.pieces[self.player].remove((x, y))
                self.pieces[self.player].append((i, j))

            # Evaluate the board state using minimax
            score = self.minimax(depth - 1, is_maximizing=False, alpha=float('-inf'),
beta=float('inf'))

            # Undo changes
            if move_type == "placement":
                self.board[x][y] = ' '
            else:
                self.board[x][y], self.board[i][j] = self.player, ' '
                self.pieces[self.player].remove((i, j))
                self.pieces[self.player].append((x, y))

            # Update the best action if a better score is found
            if score > best_score:
                best_score = score
                best_action = action

        return best_action
```

La fonction `best_move` applique l'algorithme Minimax avec élagage alpha-bêta pour identifier le meilleur déplacement possible pour l'IA. Elle fonctionne de la manière suivante :

1. **Initialisation des variables** :
    - `best_score` : Stocke le meilleur score trouvé jusqu'à présent (initialisé à -inf).
    - `best_action` : Stocke l'action correspondant au meilleur score.

2. **Génération des actions possibles** :
    - Si le bot est en phase de placement, les actions incluent toutes les cases libres sur le plateau.
    - En phase de déplacement, les actions comprennent tous les mouvements valides des pions du bot vers les cases adjacentes libres.
3. **Vérification des coups immédiats gagnants** :
    - Chaque action est simulée pour vérifier si elle mène à une victoire immédiate pour le bot.
    - Si une victoire est détectée, le bot annule la simulation et retourne immédiatement l'action gagnante.
4. **Simulation et calcul du score avec Minimax** :
    - Pour chaque action possible, le bot simule l'action en modifiant temporairement le plateau et les listes de pions.
    - La fonction `minimax` est appelée pour évaluer la conséquence de cette action. La profondeur de recherche est limitée (ici, fixée à 4 pour le mode difficile).
5. **Réinitialisation après chaque simulation** :
    - Une fois le score calculé, les modifications apportées au plateau et aux pions sont annulées pour revenir à l'état initial.
6. **Mise à jour du meilleur score et de l'action** :
    - Si le score calculé est supérieur à `best_score`, ce dernier est mis à jour avec le nouveau score, et l'action correspondante est enregistrée comme étant la meilleure.

Conclusion

Grâce à l'algorithme Minimax avec élagage alpha-bêta, la fonction `best_move` permet au bot de :

- Explorer les conséquences des actions jusqu'à une profondeur de 4 (dans le mode difficile).
- Utiliser des heuristiques pour évaluer les situations lorsque la profondeur maximale est atteinte.
- Ignorer les branches de l'arbre de recherche qui ne peuvent pas influer sur le résultat final, améliorant ainsi l'efficacité.