Report Pa0A

Rapport de IA41

Produit par :

Yann Comor

Numidia NIMHAOULIN

Guillaume STRABACH

Sous la supervision de :

Prof. F. LAURI

UTBM

5 janvier 2025

Résumé

Le but de ce rapport est de présenter le travail réalisé durant le Semestre A24 dans l'UE IA41.

Dans le cadre de cet UV, nous devions en groupe de trois réaliser un projet implémentant une Intelligence Artificielle pour de la résolution de jeu. Nous avons choisi de réaliser un jeu de plateau, plus spécifiquement le Teeko.

Nous avons réalisé un programme en python permettant de jouer au Teeko contre un joueur humain ou contre une Intelligence Artificielle (IA).

Mots-clefs : Python, AlphaBeta, MinMax, Tkinter, Teeko, IA

Acronymes

IA Intelligence Artificielle

Table des matières

Chapitre 1

Introduction

1.1 Description du projet

Ce projet est réalisé dans le cadre de l'unité d'enseignement *IA41* de l'Université de Technologie de Belfort-Montbéliard.

Il consiste en l'implémenation d'un jeu qui doit être réalisé en *Prolog* ou en *Python* (le choix retenu dans ce projet) et doit être capable de proposer des IA de difficulté croissante.

Ce document vient compléter les trois rendus demandés : le code source accessible sur Github et la présentation orale.

Le but de ce projet est de réaliser un jeu de Teeko fonctionnel, proposant à l'utilisateur de jouer contre des IA ou contre des humains. Le jeu du Teeko est un jeu de stratégie combinatoire abstrait, inventé par John Scarne en 1952. Il se joue sur un plateau de 5x5 cases et chaque joueur dispose de 4 pions. Le but du jeu est de placer ses 4 pions de manière à former une ligne, une colonne, une diagonale ou un carré.

Au début de la partie, le plateau est vide. Les joueurs placent leurs pions tour à tour. Une fois les 8 pions placés, les joueurs peuvent déplacer un pion d'une case dans n'importe quelle direction.

Le jeu se termine lorsqu'un joueur a aligné ses 4 pions.

1.2 Objectifs

Les objectifs de ce projet sont nombreux. En premier, la compréhension et l'implémentation d'IA vue en classe. La deuxième est de faire un jeu fonctionnel et agréable à jouer. La troisième est de proposer des degrés de difficultées croissants pour les IA. Enfin, le but de ce projet était pour nous de découvrir un outil de travail collaboratif pour la première fois.

1.3 Organisation du projet

Le projet s'est organisé en plusieurs étapes :

- La réflexion autour du jeu et de ce que l'on souhaitait faire
- la répartition des taches
- la réalisation du jeu et de l'interface graphique
- la réalisation des IA
- la rédaction du rapport

Afin de pouvoir travailler au mieux en collaboration, le choix été fait de travailler sur l'outil *Github* pour la gestion du code source.

Chapitre 2

Analyse

Dans ce chapitre, nous allons définir les termes du projet, spécifier les besoins, analyser la faisabilité du projet et les risques associés.

2.1 Definitions

Intelligence Artificielle

Une Intelligence Artificielle (IA) est un programme informatique capable de simuler une forme d'intelligence humaine. Elle est capable de prendre des décisions, d'apprendre et de s'adapter à

son environnement.

MinMax

L'algorithme MinMax est une méthode de décision utilisée dans les jeux à deux joueurs, comme les échecs ou le Teeko. Il consiste à explorer toutes les possibilités de jeu pour déterminer le meilleur coup à jouer. Le principe est de maximiser le gain du joueur courant tout en minimisant celui de l'adversaire. Voici les étapes principales de l'algorithme :

1. **Génération de l'arbre de jeu** : L'arbre de jeu est construit en explorant toutes les actions possibles à partir de l'état courant jusqu'à une profondeur donnée.
2. **Évaluation des feuilles** : Chaque feuille de l'arbre est évaluée à l'aide d'une fonction d'évaluation qui estime la valeur de l'état de jeu pour le joueur courant.
3. **Propagation des valeurs** : Les valeurs des feuilles sont propagées vers le haut de l'arbre en appliquant alternativement les fonctions de maximisation et de minimisation.
4. **Choix du meilleur coup** : Le coup correspondant à la valeur maximale au niveau racine est choisi comme le meilleur coup à jouer.

L'algorithme peut être optimisé à l'aide de techniques comme l'élagage alpha-bêta, qui permet de réduire le nombre de nœuds explorés en éliminant les branches qui ne peuvent pas influencer la décision finale.

AlphaBeta

L'algorithme AlphaBeta est une optimisation de l'algorithme MinMax qui permet de réduire le nombre de nœuds explorés dans l'arbre de jeu. Il utilise deux valeurs, alpha et beta, pour élaguer les branches inutiles et ainsi accélérer le processus de décision. Voici les étapes principales de l'algorithme :

1. **Initialisation** : Les valeurs alpha et beta sont initialisées respectivement à $-\infty$ et $+\infty$.
2. **Exploration de l'arbre** : L'arbre de jeu est exploré de manière similaire à l'algorithme MinMax, mais avec des conditions supplémentaires pour l'élagage.
3. **Élagage** :
   — Si une valeur est trouvée qui est pire que la valeur actuelle de beta pour le joueur maximisant, alors cette branche est élaguée (coupure beta).
   — Si une valeur est trouvée qui est pire que la valeur actuelle de alpha pour le joueur minimisant, alors cette branche est élaguée (coupure alpha).

4. **Propagation des valeurs** : Les valeurs des feuilles sont propagées vers le haut de l'arbre en appliquant alternativement les fonctions de maximisation et de minimisation, tout en mettant à jour les valeurs alpha et beta.
5. **Choix du meilleur coup** : Le coup correspondant à la valeur maximale au niveau racine est choisi comme le meilleur coup à jouer.

L'algorithme AlphaBeta permet de réduire significativement le nombre de nœuds explorés par rapport à l'algorithme MinMax, ce qui le rend plus efficace pour les jeux avec un grand espace de recherche.

2.2 Spécification (formalisation) du problème : Jeu de Teeko

2.2.1 Rappel du contexte et problématique

Le jeu de Teeko est un jeu combinatoire abstrait, joué sur un plateau carré de 5×5 cases. L'objectif est de développer une version informatique de ce jeu, permettant à deux joueurs (humains ou intelligences artificielles) de s'affronter tout en respectant les règles originales. Le programme doit garantir une expérience utilisateur fluide et proposer des intelligences artificielles (IA) à difficulté croissante. La formalisation suivante vise à définir de manière rigoureuse les éléments nécessaires à la résolution algorithmique du problème.

2.2.2 Formalisation du problème

Modélisation mathématique et logique

Le problème peut être décrit comme une configuration d'état dans un espace discret, où chaque état représente une disposition du plateau de jeu. Les transitions entre états sont définies par les actions des joueurs (placement ou déplacement d'un pion). La formalisation repose sur les notions suivantes :

Plateau de jeu :

— **Espace des états** :

$$S = \{s : (i,j) \in \{0,1,2,3,4\} \times \{0,1,2,3,4\} \mapsto \{B, W, .\}\} \tag{2.1}$$

où $s_{i,j}$ représente l'état de la case (i,j), et B, W, ou . désignent respectivement un pion noir, un pion blanc ou une case vide.

— **État initial** : Le plateau initial est une matrice vide :

$$s_0 = \{s_{i,j} = . \, \forall i, j \in [0, 4]\} \tag{2.2}$$

— **État final** : Un état est considéré comme final si un joueur a aligné ses quatre pions selon l'une des configurations suivantes :

1. Ligne horizontale : $\exists i \, \forall j \in [j_0, j_0 + 3], \; s_{i,j} = B$ ou $s_{i,j} = W$.
2. Ligne verticale : $\exists j \, \forall i \in [i_0, i_0 + 3], \; s_{i,j} = B$ ou $s_{i,j} = W$.
3. Diagonale : $\forall k \in [0, 3], \; s_{i+k,j+k} = B$ ou $s_{i+k,j+k} = W$.
4. Carré : $\exists i, j \in [0, 3], \; \big(\forall k \in \{i, i+1\}, \; \forall l \in \{j, j+1\}, \; s_{k,l} = B\big)$ ou $\big(\forall k \in \{i, i+1\}, \; \forall l \in \{j, j+1\}, \; s_{k,l} = W\big)$.

Joueurs :

— Deux joueurs, notés P_1 et P_2, alternent les tours.
— Chaque joueur possède un ensemble de pions $P = \{p_1, p_2, p_3, p_4\}$.
— L'état courant du joueur est indiqué par une variable `turn`, prenant les valeurs B ou W.

Transitions entre états : Les transitions entre états sont déterminées par les actions suivantes :

— **Placement :** Ajout d'un pion sur une case vide durant la phase de placement.
— **Déplacement :** Déplacement d'un pion vers une case adjacente vide durant la phase de mouvement.

La fonction de transition T est définie comme suit :

$$T(s, a) = s' \quad \text{où} \quad a \in A(s) \tag{2.3}$$

où $A(s)$ est l'ensemble des actions valides dans l'état s.

Fonction objectif : L'objectif est de déterminer une séquence d'actions $(a_1, a_2, \ldots, a_n)$ qui mène à un état final s_f tel que :

$$G(s_f) = \text{true} \tag{2.4}$$

où G est une fonction booléenne vérifiant les conditions de victoire.

Contraintes et règles

1. Une case ne peut contenir qu'un seul pion.
2. Un pion ne peut être déplacé que vers une case adjacente.
3. Les joueurs doivent respecter les phases de jeu :

— **Phase de placement** : Tous les pions doivent être placés avant de pouvoir se déplacer.
— **Phase de déplacement** : Les pions déjà placés peuvent être déplacés vers une case vide.

4. Le jeu se termine dès qu'une condition de victoire est remplie.

Algorithmes de vérification des règles

— Vérification de la validité d'une action (`_is_valid_cell`, `_is_adjacent`).
— Détection des conditions de victoire (`_check_line_win`, `_check_square_win`).
— Gestion des erreurs (`InvalidMoveException`).

2.2.3 Interactions avec les joueurs et l'IA

— Les joueurs interagissent via une interface graphique (GUI) implémentée dans `teeko_gui.py`.
— Les IA utilisent une copie de l'état du jeu (`copy_game()`) pour effectuer leurs calculs sans modifier l'état original.

2.3 Faisabilité et risques

2.3.1 Faisabilité

Aspects techniques

La réalisation du jeu de Teeko en Python est techniquement réalisable grâce à plusieurs raisons :

— **Gestion des règles de jeu** :
  — Les règles du jeu sont relativement simples, avec des conditions bien définies (placement, déplacement, et victoire).
— **Interface utilisateur (GUI)** :
  — L'utilisation de la bibliothèque `tkinter` permet de créer une interface graphique intuitive et fonctionnelle.
  — Les composants graphiques (`GameBoard`, `Player`, etc.) facilitent la visualisation du plateau et des mouvements des joueurs.
— **Interactions avec les IA** :
  — Le jeu est conçu pour intégrer des IA grâce à des fonctions comme `_play_ai`, qui utilise la méthode `next_move()`.
  — Le mécanisme de copie d'état (`copy_game()`) garantit que les IA peuvent simuler leurs décisions sans affecter l'état réel du jeu.
— **Modularité du code** :

— Le code est structuré en modules indépendants (`teeko.py`, `teeko_gui.py`, etc.), ce qui simplifie l'ajout de fonctionnalités ou les modifications futures.

Contraintes de développement

— **Temps disponible** :
  — Le projet est limité par la durée de l'unité d'enseignement, ce qui nécessite une planification rigoureuse.
— **Compétences techniques** :
  — Les connaissances nécessaires incluent Python, `tkinter` pour la GUI, et des concepts d'algorithmes pour les IA.
— **Tests et validation** :
  — Tester toutes les configurations possibles du jeu (placements, déplacements, conditions de victoire) peut être complexe.

2.3.2 Risques

Risques techniques

— **Complexité des IA** :
  — La conception d'IA de niveaux de difficulté croissants peut s'avérer difficile, notamment si des heuristiques avancés sont à développer.
  — Une IA trop lente peut réduire l'expérience utilisateur.
— **Gestion des erreurs** :
  — Les exceptions (`InvalidMoveException`, `InvalidTypeException`) doivent être correctement gérées pour éviter des comportements inattendus.
— **Interface graphique** :
  — Des limitations dans `tkinter` pourraient rendre difficile l'ajout de fonctionnalités avancées (par exemple, animations ou mises en évidence complexes).

Risques organisationnels

— **Collaboration** :
  — Une mauvaise coordination entre les membres de l'équipe pourrait ralentir le développement.
  — Le choix d'un outil de collaboration (comme GitHub) nécessite une bonne maîtrise des workflows Git.

Risques liés à l'utilisation

- **Satisfaction utilisateur** :
  - Une interface peu intuitive ou des IA mal calibrées pourraient réduire l'intérêt du jeu pour les utilisateurs.
- **Bugs imprévus** :
  - Des erreurs non détectées pourraient compromettre l'expérience utilisateur, notamment lors des interactions entre humains et IA.

Chapitre 3

Conception

3.1 Architecture

Le projet est conçu selon une architecture modulaire, où chaque composant joue un rôle distinct mais interconnecté. Au cœur du système, le module `teeko.py` encapsule la logique fondamentale du jeu. Il définit les règles, gère les mouvements des pions et détermine les conditions de victoire. Autour de ce noyau central gravitent d'autres modules, chacun ayant une responsabilité spécifique.

L'interface utilisateur, implémentée dans `teeko_gui.py`, offre une représentation visuelle intuitive du jeu, permettant aux joueurs d'interagir avec le plateau. Ce module s'appuie sur la bibliothèque `tkinter` pour afficher les pions, mettre en évidence les déplacements autorisés et signaler les erreurs éventuelles.

Les intelligences artificielles, quant à elles, sont organisées dans des modules spécialisés. Le fichier `BaseIA.py` définit une classe abstraite qui sert de fondation commune aux différentes stratégies, notamment celles basées sur les algorithmes MinMax et Alpha-Beta pruning. Ces algorithmes, décrits dans `MinMax.py` et `AlphaBeta.py`, permettent aux IA de planifier leurs mouvements en explorant les différentes configurations possibles du plateau.

Cette séparation des préoccupations rend le projet à la fois flexible et extensible. Chaque composant peut évoluer indépendamment, facilitant ainsi les ajouts futurs ou les modifications nécessaires.

3.2 Modèle de solution

Le jeu de Teeko repose sur un modèle combinatoire où chaque état du plateau représente une disposition spécifique des pions. Les états sont modifiés par des actions, qu'il s'agisse de placer un pion durant la phase initiale ou de le déplacer une fois cette phase terminée.
La stratégie adoptée pour résoudre le problème repose sur une exploration systématique de ces états. Les algorithmes MinMax et Alpha-Beta sont utilisés pour évaluer les configurations possibles, en tenant compte des règles du jeu et des positions des pions. Les fonctions heuristiques, décritent dans la partie 4.4, jouent un rôle clé dans cette évaluation en attribuant un score à chaque situation.
Dans ce modèle, chaque joueur agit comme un agent rationnel cherchant à maximiser ses chances de victoire tout en minimisant celles de son adversaire. L'état du jeu évolue donc selon une alternance de décisions calculées, reflétant la nature stratégique de Teeko.

3.3 UX/UI

L'expérience utilisateur a été conçue pour être fluide et intuitive. L'interface graphique, créée avec `tkinter`, affiche un plateau clair et bien structuré, où les pions noirs et blancs sont facilement distinguables. Les joueurs interagissent avec le jeu en cliquant sur les cases du plateau, les mouvements autorisés étant visuellement mis en évidence.
Des messages explicites guident l'utilisateur, signalant par exemple les erreurs en cas de coup invalide. Cette approche vise à réduire la courbe d'apprentissage et à garantir une expérience agréable, que le joueur soit débutant ou expérimenté.
L'interface ne se limite pas à une simple représentation visuelle. Elle joue également un rôle dans la dynamique du jeu, en affichant le joueur actif, le temps restant pour chaque tour et, finalement, en annonçant le vainqueur de manière claire et engageante.

3.4 Héritage et polymorphisme

L'implémentation des intelligences artificielles repose sur les concepts d'héritage et de polymorphisme, qui permettent de structurer le code de manière élégante et modulaire. La classe abstraite `BaseAI`, définie dans `BaseIA.py`, sert de socle commun à toutes les IA. Elle contient les méthodes génériques nécessaires à l'exploration des états du jeu, comme `get_all_possible_moves`, qui génère les coups autorisés en fonction de l'état actuel du plateau.
Les classes `MinMax` et `AlphaBeta` héritent de `BaseAI`, chacune implémentant sa propre stratégie pour évaluer les mouvements. `MinMax` applique un algorithme récursif classique, explorant

l'arbre des possibles jusqu'à une profondeur donnée. `AlphaBeta`, quant à elle, optimise ce processus en éliminant les branches inutiles grâce aux bornes alpha et beta.
Grâce au polymorphisme, les IA peuvent être utilisées de manière interchangeable. Le reste du code n'a pas besoin de savoir si une IA utilise MinMax ou Alpha-Beta : il lui suffit d'appeler la méthode `next_move`, définie dans `BaseAI` et redéfinie dans chaque classe dérivée.
De plus, grace à l'utilisation de l'Héritage, les classes filles peuvent redéfinir les méthodes de la classe mère pour les adapter à leur propre fonctionnement. Cela permet de réduire la duplication de code et de garantir une cohérence dans l'ensemble du projet.

3.5 AlphaBeta et MinMax

Les algorithmes MinMax et Alpha-Beta constituent le cœur de la prise de décision des IA. Ces deux méthodes partagent un objectif commun : identifier le meilleur coup possible en évaluant les configurations du plateau.
MinMax, utilisé dans `MinMax.py`, explore toutes les possibilités jusqu'à une profondeur donnée, en alternant entre les perspectives des deux joueurs. Chaque état terminal est évalué à l'aide d'une fonction heuristique, et l'algorithme retourne le mouvement qui maximise l'avantage pour le joueur actif.
Alpha-Beta, implémenté dans `AlphaBeta.py`, suit une logique similaire, mais optimise le processus en évitant d'explorer les branches dont l'issue est clairement défavorable. Cette approche réduit considérablement le nombre d'états à analyser, permettant ainsi d'approfondir davantage l'exploration.
Ces deux algorithmes, bien que différents dans leur efficacité, sont intégrés de manière homogène au projet grâce à l'héritage. Ils illustrent la puissance des concepts algorithmiques appliqués à

un problème combinatoire, tout en garantissant une expérience de jeu stimulante face aux IA.

Chapitre 4

Implementation

4.1 Algorithmes

Les algorithmes utilisés dans ce projet, en particulier `MinMax` et `AlphaBeta`, sont au cœur de la prise de décision des intelligences artificielles. Leur implémentation repose sur des principes bien établis dans le domaine des jeux combinatoires.
L'algorithme `MinMax`, défini dans le fichier `MinMax.py`, explore l'arbre des possibilités en alternant entre les perspectives des deux joueurs. À chaque niveau de l'arbre, il simule tous les coups possibles, calcule la valeur de chaque configuration à l'aide des heuristiques, et choisit la meilleure option pour maximiser les chances de victoire ou minimiser les pertes. L'évaluation des états est réalisée uniquement lorsque l'algorithme atteint la profondeur maximale définie par `self.depth` ou lorsque le jeu est terminé.
L'algorithme `AlphaBeta`, défini dans `AlphaBeta.py`, améliore l'efficacité de `MinMax` en utilisant des bornes alpha et beta. Ces bornes permettent d'éliminer certaines branches de l'arbre des possibilités lorsque leur exploration ne peut pas influencer le résultat final. Grâce à cette optimisation, l'algorithme peut explorer des arbres plus profonds dans le même temps d'exécution, ce qui le rend particulièrement adapté aux IA plus avancées.
Ces deux algorithmes interagissent directement avec le module `teeko.py`, qui fournit les méthodes nécessaires pour simuler les mouvements (`move`) et évaluer si une configuration du plateau correspond à une fin de jeu (`is_game_over`). Ils s'appuient également sur la méthode `copy_game` pour explorer les configurations sans modifier l'état actuel du plateau.

4.2 POO

Le projet exploite les concepts fondamentaux de la programmation orientée objet (POO) pour structurer le code de manière modulaire et réutilisable. Les principaux principes utilisés sont les

suivants :

Encapsulation

Chaque module du projet encapsule des responsabilités spécifiques. Par exemple, `teeko.py` gère exclusivement les règles du jeu, tandis que `teeko_gui.py` se concentre sur l'interface utilisateur. Cette séparation garantit que les modifications apportées à un module n'affectent pas les autres.

Héritage

Le fichier `BaseIA.py` définit une classe abstraite, `BaseAI`, qui sert de base pour toutes les intelligences artificielles. Les classes `MinMax` et `AlphaBeta` héritent de cette classe et redéfinissent les méthodes nécessaires, comme `next_move`. L'utilisation de l'héritage permet de partager le code commun entre toutes les IA, tout en permettant à chaque algorithme d'avoir sa propre implémentation.

Polymorphisme

Grâce à l'héritage, les objets des classes `MinMax` et `AlphaBeta` peuvent être traités de manière uniforme. Par exemple, le reste du code n'a pas besoin de savoir quel algorithme est utilisé : il lui suffit d'appeler la méthode `next_move`, qui est définie dans `BaseAI` et redéfinie dans chaque sous-classe. Ce polymorphisme renforce la modularité et la flexibilité du projet.

Abstraction

La classe `BaseAI` utilise le module `ABC` pour définir des méthodes abstraites, comme `next_move`, qui doivent obligatoirement être implémentées dans les classes dérivées. Cela garantit que toutes les IA respectent une interface commune.

4.3 Gestion des erreurs

La gestion des erreurs est une composante essentielle du projet, assurant que les comportements inattendus soient correctement signalés sans interrompre l'exécution du programme.
Deux exceptions spécifiques sont définies dans le fichier `teeko.py` :

— `InvalidMoveException` : Lancée lorsque le joueur tente un mouvement invalide, par exemple en plaçant un pion sur une case occupée ou en déplaçant un pion sur une case non adjacente.

— `InvalidTypeException` : Utilisée pour signaler des erreurs de type, bien qu'elle soit moins fréquemment rencontrée.

Ces exceptions sont capturées dans les modules appelants, notamment dans `teeko_gui.py`, où des messages explicites sont affichés à l'utilisateur pour indiquer la nature de l'erreur. Par exemple, si un joueur tente de placer un pion dans une case déjà occupée, un message est affiché dans l'interface pour expliquer pourquoi l'action est invalide.

Du côté des intelligences artificielles, la gestion des erreurs est intégrée directement dans les algorithmes. Avant d'appliquer un mouvement simulé, les méthodes `apply_move` et `get_all_possible_moves` vérifient que le coup est valide en s'appuyant sur les règles du jeu définies dans `teeko.py`.

Enfin, la robustesse du projet est renforcée par des vérifications systématiques de la validité des actions. Ces mécanismes garantissent que l'état du jeu reste cohérent, même en cas d'entrée incorrecte ou de scénario inattendu.

4.4 Heuristiques

Les heuristiques jouent un rôle crucial dans la prise de décision des intelligences artificielles (IA) implémentées pour le jeu Teeko. Elles permettent d'évaluer la qualité d'un état donné du plateau, en attribuant des scores basés sur des critères stratégiques. Ces scores orientent les algorithmes MinMax et Alpha-Beta dans leur exploration des coups possibles. Voici une présentation des principales heuristiques utilisées et de leur fonctionnement.

Contrôle central (`evaluate_central_control`)

Le contrôle du centre est une stratégie fondamentale dans de nombreux jeux de plateau. Cette heuristique évalue la capacité du joueur à occuper les cases proches du centre, qui offrent généralement une meilleure mobilité et un contrôle accru du jeu.

Fonctionnement : Pour chaque pion du joueur, la distance par rapport au centre du plateau est calculée. Les positions proches du centre reçoivent un score plus élevé, tandis que celles situées en périphérie sont moins valorisées. La formule suivante est utilisée :

$$\text{score} = \max(0, 3 - \text{distance au centre})$$

Avantages :

— Favorise une position stratégique, permettant de réagir efficacement aux mouvements adverses.

— Encourage un jeu équilibré et adaptable.

Limites :

— Moins pertinent en fin de partie, lorsque l'action se déplace souvent loin du centre.
— Néglige les positions périphériques stratégiques dans certains cas.

Mobilité (`evaluate_mobility`)

La mobilité évalue la flexibilité d'un joueur en termes de mouvements possibles pour ses pions. Plus un joueur a d'options, plus il est considéré comme étant dans une position avantageuse.
Fonctionnement : Pour chaque pion, le nombre de cases adjacentes libres est compté. La somme de ces mouvements possibles constitue le score de mobilité. Cette évaluation favorise les configurations dynamiques et empêche l'enfermement des pions.
Avantages :
— Encourage un jeu actif et adaptable.
— Permet de contrer les positions défensives statiques.
Limites :
— Peut ignorer les alignements stratégiques.
— Trop favoriser la mobilité peut détourner l'attention des menaces immédiates.

Proximité de victoire (`evaluate_near_victory`)

Cette heuristique identifie les configurations proches de la victoire, par exemple, trois pions alignés avec une case libre pour compléter une ligne de quatre.
Fonctionnement : Le plateau est analysé pour détecter les alignements (lignes, colonnes, diagonales) ou carrés 2×2 qui sont à une pièce de compléter une victoire. Chaque configuration détectée reçoit un score élevé, ce qui pousse l'IA à exploiter rapidement les opportunités gagnantes.
Avantages :
— Met l'accent sur les opportunités offensives cruciales.
— Oriente l'IA vers des mouvements décisifs.
Limites :
— Néglige les contre-mesures défensives si utilisée seule.
— Peut surévaluer des configurations difficiles à concrétiser.

Blocage des adversaires (`evaluate_block_opponent`)

Pour contrer les menaces adverses, cette heuristique identifie les configurations où l'adversaire est proche de la victoire et favorise les coups qui bloquent ces opportunités.
Fonctionnement : Les alignements ou carrés 2×2 proches de la victoire pour l'adversaire sont détectés et pénalisés. L'IA attribue un score négatif élevé à ces configurations, ce qui la pousse

à intervenir pour empêcher une victoire rapide.

Avantages :

- Renforce la défense de l'IA en anticipant les menaces adverses.
- Permet de maintenir une position équilibrée tout en jouant de manière proactive.

Limites :

- Une trop grande priorité donnée à cette heuristique peut détourner l'IA de ses propres opportunités offensives.
- Moins efficace en début de partie, lorsque les menaces sont encore faibles.

Pondération et combinaison des heuristiques

Pour obtenir un comportement équilibré, les heuristiques sont pondérées en fonction de leur importance relative dans le jeu. Ces pondérations vont varier d'une IA à l'autre, de même que les heuristiques utilisées. Les pondérations suivantes ont par exemple été utilisées :

$$\text{Score total} = 2 \cdot \text{contrôle central} + 5.5 \cdot \text{mobilité} + 15 \cdot \text{proximité de victoire} + 10 \cdot \text{blocage de l'adversaire}$$

Cette combinaison favorise une logique offensive tout en assurant une défense efficace, garantissant un jeu stratégique à toutes les phases.

Heuristiques abandonnées

Au cours du développement de l'IA pour le jeu Teeko, plusieurs heuristiques ont été conçues et testées. Bien que certaines aient montré un potentiel initial, elles ont finalement été abandonnées en raison de leur manque de pertinence stratégique, de leur complexité computationnelle excessive, ou d'une faible contribution à la victoire. Voici quelques exemples de ces heuristiques :

Distance cumulative entre pions alliés Cette heuristique évaluait la proximité des pions d'un même joueur, en attribuant un score plus élevé aux configurations où les pions étaient regroupés. L'objectif était de favoriser des formations compactes, supposées plus faciles à défendre et à exploiter offensivement.
Cependant, bien que cette heuristique puisse être utile dans certains scénarios, elle s'est révélée problématique dans des configurations où la dispersion des pions était stratégiquement nécessaire, notamment pour bloquer l'adversaire ou contrôler différentes zones du plateau. De plus, le calcul de toutes les distances cumulatives augmentait significativement le temps de calcul de l'IA, rendant cette méthode impraticable pour une expérience de jeu fluide.

Évaluation de la symétrie du plateau Une autre approche visait à analyser la symétrie des

positions des pions sur le plateau. L'hypothèse était que des dispositions symétriques pourraient faciliter l'adaptation stratégique, en offrant des options équivalentes sur plusieurs axes.
Malgré son élégance conceptuelle, cette heuristique n'a apporté aucun avantage concret dans les parties testées. Les configurations symétriques n'ont pas contribué directement à des victoires ou à des blocages efficaces, et leur calcul était inutilement complexe, nécessitant une analyse détaillée de l'état global du plateau.

Prévention des doubles alignements Cette heuristique tentait d'identifier et de bloquer les configurations où l'adversaire pouvait menacer simultanément deux alignements potentiels. Bien qu'elle ait montré une certaine efficacité dans des scénarios spécifiques, sa complexité résidait dans la détection de tous les "chemins croisés" possibles entre les lignes, colonnes et diagonales.
Le calcul systématique de ces menaces croisées consommait un temps de calcul disproportionné par rapport aux gains stratégiques obtenus. En pratique, cette heuristique a souvent doublé ou triplé le temps de réponse de l'IA, sans améliorer significativement son niveau de jeu.

Conclusion

Au cours du projet, l'essentiel du travail s'est donc porté sur l'implémentation des heuristiques, car ce sont elles qui vont déterminer la qualité de notre IA. Plusieurs pistes ont été développées, certaines concervées, d'autres abandonnées.
L'objectif final était néanmoins de travailler sur des combinaisons de ces heuristiques, en les pondérants. Ainsi, le fait de regrouper toutes les heuristiques dans le fichier `heuristics.py` permet de les appeler facilement dans les classes `MinMax` et `AlphaBeta`, et de les combiner pour obtenir un score global tout en évitant une redondance de code.

Chapitre 5

Résultats

5.1 Résultats visuels

Le développement du jeu Teeko a permis de concevoir une interface utilisateur visuellement claire et fonctionnelle, comme illustré par les images suivantes. Chaque phase du jeu est représentée graphiquement, offrant une expérience utilisateur fluide et intuitive.

Plateau vide

En premier, le plateau initialement vide est présenté pour marquer le début de chaque partie.

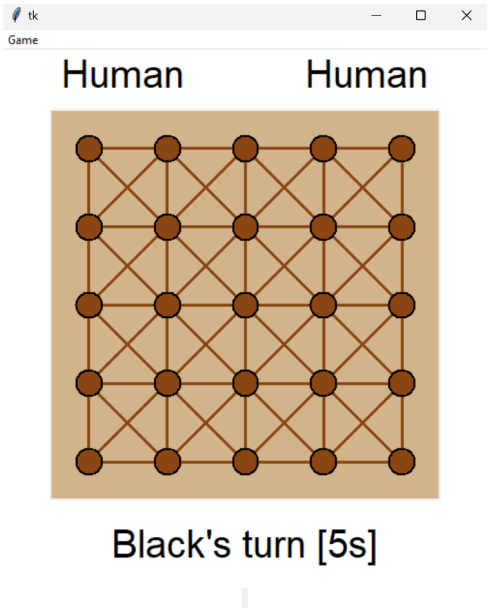

FIGURE 5.1 – Plateau vide au début d'une partie.

Écran de sélection

L'écran de sélection permet au joueur de choisir le mode de jeu, qu'il s'agisse d'un affrontement humain contre humain ou humain contre IA, avec différents niveaux de difficulté. Cet écran sert d'introduction visuelle au jeu.

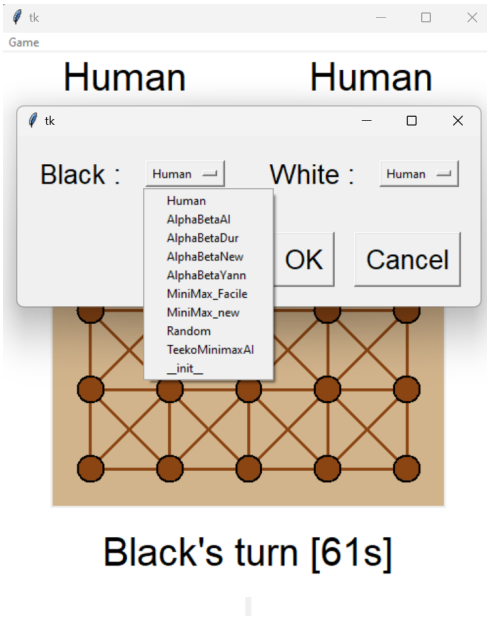

FIGURE 5.2 – Écran de sélection des joueurs et du mode de jeu.

Phase de placement

La phase de placement est illustrée par un plateau vide. Les joueurs ajoutent leurs pions en alternance sur des cases libres.

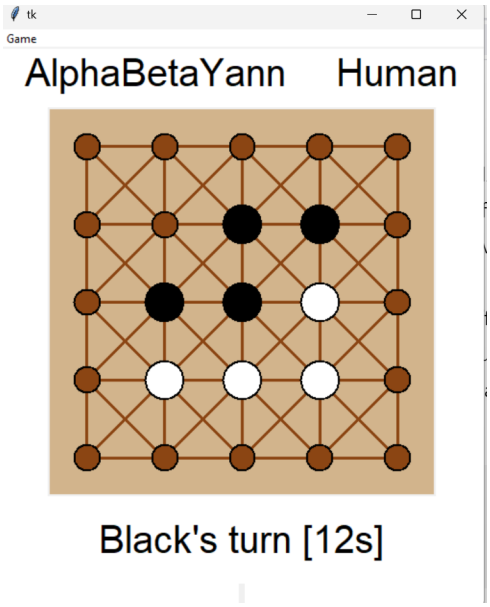

FIGURE 5.3 – Phase de placement : les joueurs positionnent leurs pions.

Phase de déplacement

Une fois tous les pions placés, le jeu passe à la phase de déplacement. Les joueurs peuvent déplacer leurs pions sur des cases adjacentes, comme indiqué ci-dessous.

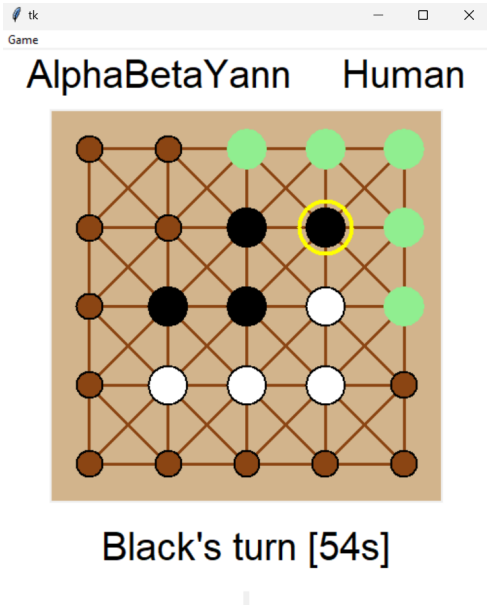

FIGURE 5.4 – Phase de déplacement : les joueurs déplacent leurs pions.

Messages d'erreur

Lorsqu'un mouvement invalide est tenté, un message d'erreur clair est affiché pour guider l'utilisateur.

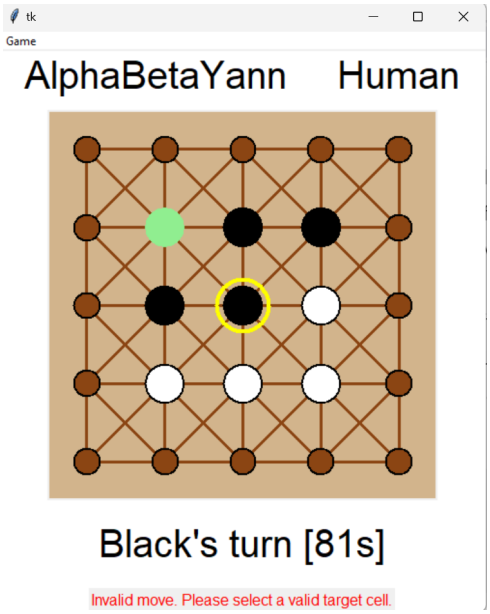

FIGURE 5.5 – Message d'erreur affiché en cas de mouvement invalide.

Victoire

Enfin, en cas de victoire d'un joueur, un message de félicitations est affiché pour annoncer le gagnant. En plus de cela, le temps de jeu pour chaque joueur est affiché.

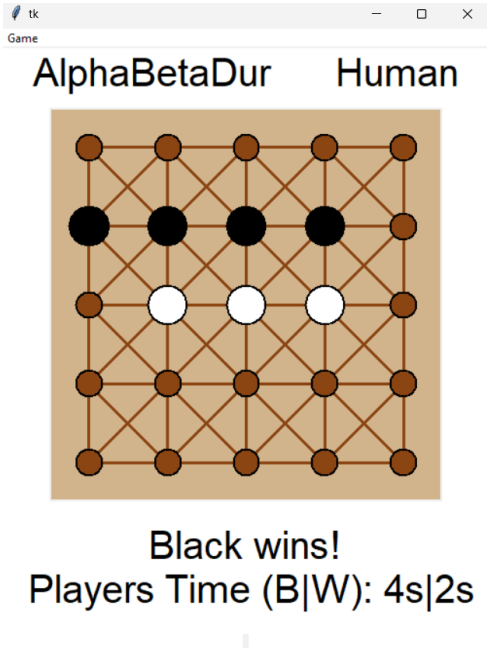

FIGURE 5.6 – Message de victoire affiché à la fin de la partie.

5.2 Résultats des programmes

Les résultats des programmes démontrent que l'IA est capable de jouer à différents niveaux de difficulté tout en respectant les règles du jeu. Voici quelques observations principales :

- Les IA utilisant l'algorithme MinMax montrent une efficacité acceptable, mais leur performance diminue lorsque la profondeur de recherche est limitée. De plus, augmenter la profondeur rend le temps de calcul prohibitif (ce qui est cohérent).
- Les IA basées sur Alpha-Beta pruning offrent des performances supérieures grâce à une optimisation du temps de calcul, permettant une exploration plus profonde.
- Les heuristiques implémentées, comme le contrôle central et la mobilité, ont permis d'améliorer la compétitivité de l'IA, bien qu'une attention particulière soit nécessaire pour équilibrer les pondérations.

Néanmoins, certains points plus négatifs ont été identifiés, notamment :

- Les performances de l'IA sont limitées par les ressources matérielles, en particulier pour des profondeurs de recherche élevées.
- L'optimisation des heuristiques reste un défi, car des ajustements inappropriés peuvent entraîner des comportements inattendus.
- L'augmentation des heuristiques ne conduit pas forcément à une augmentation de la qualité des IA.

Par exemple, pour les IA utilisant l'algorithme MinMax, la profondeur de recherche a été limitée à 3 pour des raisons de performance. Cette profondeur ne permet pas forcément d'empêcher les

coups gagnants pour l'adversaire, ce qui peut donner l'impression que l'IA est moins compétente. Pour y remédier, un ajout (qui a été fait dans la fonction `min_max`) permet de vérifier directement si le prochain coup peut être gagnant et agir en conséquence sans appeler `evaluate_board`. Cette manière de faire assure une meilleure défense de l'IA que lorsqu'on utilise uniquement les heuristiques.
Cependant, lorsque le jeu est dans une configuration dans laquelle il est impossible pour l'IA d'empêcher la victoire de l'adversaire, un message d'erreur s'affiche dans la console et l'IA ne joue pas.
Par exemple, dans la capture d'écran 5.7, l'humain joue les pièces noires et l'IA joue les pièces blanches. Au prochain coup, les noirs sont sûrs de gagner car il est impossible pour les blancs de les bloquer. L'IA s'est désactivée et c'est au joueur de jouer pour elle. Ensuite, il peut jouer sa pièce noire dans le coin en bas à droite et gagner.

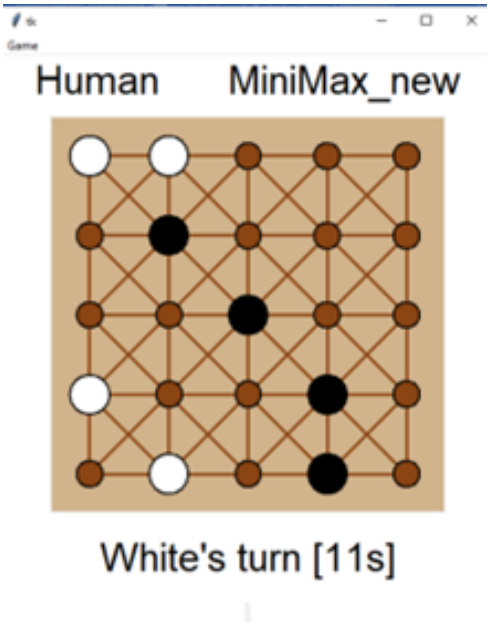

FIGURE 5.7 – Problème de configuration du plateau.

Cela reflète bien la faiblesse de l'IA concernant l'anticipation des coups.

5.3 Matériel de tests

Les tests finaux ont été effectués sur une machine disposant des caractéristiques suivantes (code fourni en annexe 7.1) :

- Processeur : AMD64 Family 23 Model 96 Stepping 1, AuthenticAMD (8 cœurs, 16 threads), 0.0 MHz de base, 2900.0 MHz en mode turbo.
- Mémoire vive : 15 Go DDR4.

— Système d'exploitation : Windows 11, version 10.0.22631.
— Environnement de développement : Python 3.12.8 avec les bibliothèques `tkinter`.

5.4 Problème minuteur

Au cours du développement, un problème a été identifié avec la gestion du minuteur lorsque l'IA prenait un temps de calcul important pour jouer son coup. Ce temps de calcul était injustement ajouté au temps du joueur suivant, ce qui faussait le décompte du temps disponible. Ce dysfonctionnement impactait l'équité du jeu et nécessitait une solution.

Solutions envisagées

Pour résoudre ce problème, deux solutions principales ont été étudiées :

Solution 1 : Ajustement manuel du temps (implémentée) La première solution, qui a été retenue et implémentée, consiste à ajuster le minuteur en faveur du joueur humain lorsque c'est une IA qui joue. Concrètement, le temps utilisé par l'IA est directement transféré au joueur suivant, de manière à compenser le temps de calcul long de l'IA.
Cette approche a l'avantage d'être simple à implémenter et de résoudre immédiatement le problème. Elle s'appuie sur une logique pragmatique : en considérant que l'IA n'a pas de limite de réflexion, son temps de calcul peut être neutralisé sans pénaliser le joueur humain. Cette solution est cependant une solution temporaire, car elle contourne le problème au lieu de le résoudre structurellement.

Solution 2 : Utilisation du multithreading (non implémentée) Une alternative plus robuste aurait été de recourir au multithreading. Cette approche consiste à exécuter le minuteur dans un thread distinct, indépendant du fil principal où l'IA effectue ses calculs. De cette manière, le minuteur fonctionne de manière asynchrone, sans être impacté par le temps de calcul de l'IA.
Bien que conceptuellement plus élégante, cette solution a été écartée pour des raisons pratiques. Elle aurait nécessité une gestion complexe des ressources, notamment pour synchroniser les threads et éviter des conflits d'accès aux variables partagées. De plus, le Global Interpreter Lock (*GIL*) de Python limite l'efficacité du multithreading dans certains contextes, ce qui aurait pu compliquer davantage l'implémentation (plus d'information dans cet article).

Résultats

La solution implémentée a permis de contourner efficacement le problème. Le transfert du temps de réflexion de l'IA au joueur suivant garantit une expérience de jeu équilibrée, sans

ralentissement notable ou confusion dans le décompte du temps.
Cette approche pragmatique a également réduit les risques liés à des modifications structurelles plus complexes, tout en maintenant la fluidité de l'expérience utilisateur.

Chapitre 6

Conclusion, limites et perspectives

6.1 Difficultés rencontrées

Le développement de ce projet a été marqué par plusieurs défis techniques et organisationnels. Tout d'abord, la gestion des algorithmes d'intelligence artificielle, en particulier leur optimisation à l'aide d'heuristiques, a nécessité un travail d'expérimentation et de calibration, travail pas toujours concluant. Trouver un équilibre entre performance et temps de calcul a souvent été un processus itératif.

Ensuite, l'interface utilisateur (*GUI*) basée sur `tkinter` a posé des contraintes en termes de gestion des interactions en temps réel, notamment avec l'ajout du minuteur. La nécessité d'assurer une synchronisation fluide entre le jeu et le chronomètre a révélé les limites du modèle monothread de Python, obligeant à adopter des solutions simplifiées pour garantir la stabilité du programme.

Enfin, la collaboration sur un projet modulaire et structuré a demandé une organisation stricte, en particulier pour intégrer les différentes parties (IA, règles du jeu, interface graphique) dans un tout cohérent. L'utilisation de GitHub a permis de centraliser les efforts, bien que la gestion des conflits de code ait parfois ralenti le développement.

6.2 Limites

Malgré les efforts déployés, certaines limitations persistent dans le projet :

Temps de calcul des IA

Les IA utilisant les algorithmes MinMax et Alpha-Beta sont performantes pour des profondeurs limitées, mais elles deviennent rapidement trop lentes à mesure que la profondeur augmente. Cela peut rendre le jeu moins fluide, notamment face à des niveaux de difficulté plus élevés. Une solution envisageable serait l'intégration d'algorithmes plus avancés, comme les Monte Carlo Tree Search (MCTS).

Gestion du minuteur

Bien que fonctionnelle, la solution implémentée pour le minuteur repose sur une logique ad hoc qui transfère le temps de calcul de l'IA au joueur humain suivant. Comme mentionné précédemment, une gestion indépendante via le multithreading ou le multiprocessing offrirait une solution plus robuste et élégante, mais elle n'a pas été retenue dans le cadre de ce projet pour des raisons de temps et de complexité.

Interface utilisateur

L'utilisation de `tkinter` a permis de concevoir une interface simple et fonctionnelle. Toutefois, cette bibliothèque présente des limites en termes de flexibilité et de personnalisation graphique. Une interface plus moderne, basée sur des frameworks comme `PyQt` ou `Kivy`, pourrait améliorer l'expérience utilisateur.

Calibrage des heuristiques

Bien que les heuristiques retenues offrent un bon équilibre entre stratégie offensive et défensive, leur calibration reste perfectible.
De plus, malgré tout nos efforts, nous n'avons pas réussi à créer une IA vraiment forte autant contre un humain que contre d'autres IA. Nos heuristiques, bien que pertinentes, ne sont sans doute pas les meilleures ce qui explique en partie ce manque de performance.

6.3 Perspectives

Les perspectives pour ce projet sont multiples et offrent de nombreuses opportunités d'amélioration :

— **Optimisation des IA** : Intégrer des algorithmes plus avancés, comme MCTS ou un Renforced Learning, pour réduire le temps de calcul tout en maintenant un niveau de jeu compétitif.

— **Gestion des processus** : Remplacer la solution actuelle du minuteur par une gestion basée sur des processus indépendants, permettant une meilleure répartition des tâches et une fluidité accrue.
— **Interface graphique améliorée** : Migrer vers un framework plus moderne pour offrir une expérience utilisateur plus attractive et interactive, avec des animations et des fonctionnalités supplémentaires.
— **Mode multijoueur en ligne** : Étendre le jeu pour permettre des parties multijoueurs en ligne via un serveur centralisé. (or cadre de ce cours)
— **Analyse des parties** : Ajouter une fonctionnalité d'analyse post-match pour fournir des statistiques et des recommandations aux joueurs, rendant le jeu éducatif et compétitif.

6.4 Conclusion

Le projet Teeko a permis de concevoir une solution fonctionnelle et ludique pour jouer à ce jeu de plateau, en mettant en oeuvre les algorithmes et concepts vus en cours d'IA41. Bien que certaines limitations subsistent, elles ouvrent la voie à des améliorations futures, tant sur le plan technique qu'ergonomique. Ce projet a constitué une opportunité enrichissante pour approfondir les concepts de programmation orientée objet, d'IA et de gestion de projet collaboratif.
En effet, ce projet a été pour certains membres du groupe une découverte de l'implémentation d'IA ou de l'outil qu'est GitHub. Il a donc été très formateur et a permis de mettre en pratique les connaissances acquises en cours, tout en se préparant à la vie professionnelle.

Chapitre 7

Annexes

7.1 Information du système

```
import platform
import psutil

def get_system_info():
    processor = platform.processor()
    cpu_info = f"{processor} ({psutil.cpu_count(logical=False)} cœurs,
{psutil.cpu_count(logical=True)} threads)"

    try:
        freq = psutil.cpu_freq()
        if freq:
            cpu_info += f", {freq.min:.1f} MHz de base, {freq.max:.1f} MHz en mode
turbo"
    except Exception:
        pass

    ram = psutil.virtual_memory().total / (1024 ** 3)
    ram_info = f"{int(ram)} Go DDR4"

    os_info = f"{platform.system()} {platform.release()}, version
{platform.version()}"

    python_info = f"Python {platform.python_version()}"


    return {
        "Processeur": cpu_info,
        "Mémoire vive": ram_info,
        "Système d’exploitation": os_info,
        "Environnement de développement": f"{python_info}"
    }

info = get_system_info()

print("Les tests finaux ont été effectués sur une machine disposant des
caractéristiques suivantes :\n")
for key, value in info.items():
    print(f"- {key} : {value}")
```