Université Abdelmalek Essaadi
Faculté des sciences et techniques de Tanger
Département génie informatique
Cycle d'ingénieur Logiciels et systèmes intelligents « LSI »
Année universitaire 2023/2024

Rapport

Mini-projet de IA : Mancala

Réalisé par:

Fatima Zahra Ait Hssaine
Safae Hammouch

Encadré par:

Pr. M'hamed AIT KBIR

Année universitaire : 2024-2025

Table de Matières

Introduction

Le projet **Mancala** est une implémentation numérique d'un jeu de société stratégique classique. Il propose un mode **Humain vs Humain** ainsi qu'un mode **Humain vs Ordinateur** assisté par intelligence artificielle. L'algorithme Alpha-Beta est utilisé pour rendre les décisions de l'ordinateur efficaces et optimales. Le projet met en œuvre une interface utilisateur intuitive à l'aide de Java Swing.

1.Objectifs principaux :

- ⇨ Créer une expérience utilisateur engageante et conforme aux règles traditionnelles de Mancala.
- ⇨ Implémenter une intelligence artificielle adaptable en fonction des niveaux de difficulté.
- ⇨ Permettre aux joueurs de sauvegarder, charger des parties pour plus de flexibilité et de demander de l'aide trois fois.

2.Technologies et outils utilisés :

- **Langage :** Java 17.
- **Interface utilisateur :** Swing.
- **Environnement de développement :** IntelliJ IDEA.
- **Versionnement :** Git/Github.
- **Format de sauvegarde :** Fichiers texte simples pour les parties sauvegardées.

3.Architecture du projet :

Le projet **Mancala** repose sur une architecture organisée en plusieurs classes principales, chacune ayant un rôle spécifique pour assurer le bon fonctionnement du jeu.

- ***Classes principales :***
    1. **MancalaMenuUI** : Cette classe représente l'interface utilisateur permettant aux joueurs de choisir le mode de jeu (jouer contre un autre humain ou contre la machine).

2. **MancalaGUI** : Cette classe gère l'interface utilisateur pour le mode de jeu **Human vs Human**. Elle fournit les interactions nécessaires pour que deux joueurs puissent jouer ensemble.
3. **MancalaGameUI** : Cette classe est dédiée à l'interface pour le mode **Human vs Computer**. Elle intègre des fonctionnalités spécifiques pour jouer contre une intelligence artificielle.
4. **Mancala** : Cette classe est au cœur de la gestion des règles du jeu. Elle implémente la logique de fonctionnement du jeu Mancala, notamment la répartition des graines dans les puits, les captures, et la vérification des conditions de fin de partie.
5. **MancalaPosition** : Cette classe décrit l'état actuel du plateau de jeu. Elle contient les informations nécessaires pour représenter la disposition des graines dans chaque puits et les scores des joueurs.
6. **MancalaMove** : Une sous-classe associée à MancalaPosition, qui modélise les mouvements possibles dans le jeu, en tenant compte des règles du Mancala.
7. **GameSearch** : Cette classe implémente les algorithmes de recherche nécessaires pour calculer les mouvements optimaux, notamment la stratégie **alpha-beta pruning** (taille alpha-bêta).

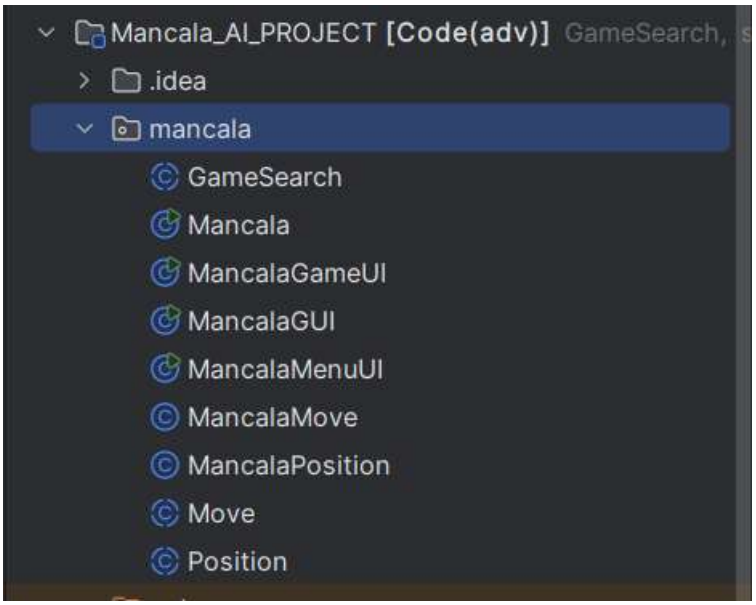

Figure 1 : Structure de projet

Fonctionnalités principales

1. Modes de jeu :

a. ***Humain vs Humain :***

Deux joueurs partagent le même appareil. Chaque joueur joue son tour en sélectionnant un pit, et les tours alternent jusqu'à la fin de la partie. La gestion des mouvements et des tours est effectuée par la méthode playGame(Position startingPosition, boolean humanPlayFirst, boolean playAgainstComputer) dans la classe GameSearch, et le paramètre playAgainstComputer est défini sur false pour ce mode.

b. ***Humain vs Ordinateur*** *:*

L'ordinateur joue contre un joueur humain. L'IA calcule son mouvement en fonction d'un niveau de difficulté sélectionné (Simple, Medium, Hard).

```
if (playAgainstComputer) { // Mode contre l'ordinateur
    System.out.println("Computer's turn!");
    List<Object> result = alphaBeta( depth: 0, pos, PROGRAM);
    MancalaPosition bestMove = (MancalaPosition) result.get(1);
    pos = bestMove;

    if (!pos.extraTurn) {
        currentPlayer = HUMAN; // Retourner à Human
    } else {
        System.out.println("Computer gets another turn!");
    }
```

La méthode **alphaBeta(int depth, Position p, boolean player)** gère l'algorithme de décision de l'ordinateur.

```
protected List<Object> alphaBetaHelper(int depth, Position p, boolean player, float alpha, float beta) {
    if (reachedMaxDepth(p, depth) || drawnPosition(p)) {
        float eval = positionEvaluation(p, player);
        return Arrays.asList(eval, null); // Score et pas de mouvement
    }
    List<Object> bestMove = new ArrayList<>();
    Position[] moves = possibleMoves(p, player);
    Arrays.sort(moves, Comparator.comparingDouble(m -> -positionEvaluation((Position) m, player)));

    float bestValue = Float.NEGATIVE_INFINITY;

    for (Position move : moves) {
        // Exploration récursive
        List<Object> evalResult = alphaBetaHelper( depth: depth + 1, move, !player, -beta, -alpha);
        float eval = -((Float) evalResult.get(0)); // Inverser l'évaluation pour l'adversaire

        if (eval > bestValue) {...}
        // Mettre à jour alpha et réaliser un cutoff si nécessaire
        alpha = Math.max(alpha, eval);
        if (alpha >= beta) {
            break;
        }
    }
    bestMove.add( index: 0, bestValue); // Ajouter le score en premier
    return bestMove;
}
// Helper function to get the pit index from the best move
private int getPitIndex(MancalaPosition current, MancalaPosition best) {  no usages
    for (int i = 0; i < 6; i++) {
        if (!Arrays.equals(current.board, best.board)) {
            return i; // Return the index of the pit where the move was made
        }
```

- ⇨ Elle implémente l'algorithme **Alpha-Beta Pruning**, une optimisation de l'algorithme Minimax utilisée pour prendre des décisions dans les jeux à deux joueurs comme Mancala. Cette méthode est utilisée par l'ordinateur pour évaluer le meilleur coup possible tout en réduisant l'arbre de recherche en éliminant des branches inutiles.
- ⇨ La configuration de la difficulté est effectuée dans la méthode **setDifficulty(Difficulty difficulty)** de la classe Mancala.

2. Niveaux de difficulté pour l'ordinateur :

a. ***Simple :***

L'ordinateur effectue des mouvements rapides sans analyser en profondeur.En plus, la profondeur maximale de recherche est limitée à 2 dans reachedMaxDepth(Position p, int depth).

b. ***Medium :***

L'ordinateur effectue une recherche en profondeur modérée pour équilibrer performance et stratégie, et la profondeur de recherche varie en fonction du nombre de graines restantes dans reachedMaxDepth(Position p, int depth).

c. ***Hard :***

L'ordinateur explore en profondeur et utilise des stratégies avancées. Une profondeur maximale élevée (jusqu'à 12) est utilisée dans reachedMaxDepth(Position p, int depth).

```
int maxDepth;
switch (difficulty) {
    case SIMPLE:
        maxDepth = 2; // Profondeur limitée pour des décisions rapides
        break;
    case MEDIUM:
        maxDepth = totalSeeds > 20 ? 6 : 8; // Exploration modérée
        break;
    case HARD:
        maxDepth = totalSeeds > 20 ? 8 : 12; // Exploration approfondie
        break;
    default:
        maxDepth = 6; // Défaut au cas où
}
```

3. Sauvegarde/Chargement :

Permet de sauvegarder l'état du plateau et de le restaurer ultérieurement.

- ***Fonctionnalités associées :***

  La méthode **saveGame(MancalaPosition position, String filename)** dans la classe Mancala sauvegarde la partie dans un fichier en utilisant un ObjectOutputStream.
  La méthode **loadGame(String filename)** restaure l'état d'un fichier en utilisant un ObjectInputStream.

- ***Options (Quitter, sauvegarder ou gérer les parties sauvegardées)*** *:*

  Affiche un menu permettant de sauvegarder, charger ou quitter la partie.

Ces options sont accessibles via l'entrée "options" dans la méthode **playGame().**

```
public void saveGame(MancalaPosition position, String filename) { 4 usages
    try (ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream(filename))) {
        out.writeObject(position);
        System.out.println("Game saved to " + filename);
    } catch (IOException e) {
        System.err.println("Failed to save game: " + e.getMessage());
    }
}
```

```
// Load the game state from a file
public MancalaPosition loadGame(String filename) {  4 usages
    try (ObjectInputStream in = new ObjectInputStream(new FileInputStream(filename))) {
        MancalaPosition position = (MancalaPosition) in.readObject();
        System.out.println("Game loaded from " + filename);
        return position;
    } catch (IOException | ClassNotFoundException e) {
        System.err.println("Failed to load game: " + e.getMessage());
        return null;
    }
}
```

4.Aide (Help) :

L'IA suggère le meilleur mouvement pour le joueur humain.

- ***Fonctionnalités associées :***

La méthode **alphaBeta()** est utilisée pour calculer le meilleur mouvement.Un compteur limite le nombre d'utilisations de l'aide (remainingHelps=3).

```
private int getBestMovePitIndex(MancalaPosition current, MancalaPosition best, boolean player) {
    int start = player ? 0 : 7;
    int end = player ? 5 : 12;

    for (int i = start; i <= end; i++) {
        if (current.board[i] > 0) { // Vérifie que le puits n'est pas vide
            MancalaPosition simulatedMove = new MancalaPosition();
            System.arraycopy(current.board,  srcPos: 0, simulatedMove.board,  destPos: 0,  length: 14);
            makeMove(simulatedMove, player, new MancalaMove(i)); // Simule le mouvement
            if (Arrays.equals(simulatedMove.board, best.board)) {
                return i; // Retourne l'indice du puits correspondant
            }
        }
```

Heuristiques implémentées

Heuristique 1 : Maximiser les graines dans le Mancala

Dans la méthode positionEvaluation, cette heuristique est réalisée par la variable mancalaDifference. Cette différence favorise les positions où le Mancala du joueur contrôlé par l'IA (PROGRAM) contient plus de graines que celui du joueur humain (HUMAN).

Rôle : Maximiser les graines dans son propre Mancala est crucial car cela contribue directement au score final du jeu. Cela guide l'évaluation pour favoriser les mouvements qui ajoutent des graines au Mancala du joueur.

```
// Différence des Mancalas
int mancalaDifference = pos.board[MancalaPosition.PROGRAM_MANCALA] - pos.board[MancalaPosition.HUMAN_MANCALA];
```

Heuristique 2 : Minimiser les graines adverses proches de leur Mancala

Dans la méthode positionEvaluation, cette heuristique est réalisée par la variable minimizeOpponentStones. Cette somme des graines dans les cases proches du Mancala adverse (cases 7 à 12 pour le joueur PROGRAM) est pénalisée.

Rôle : Réduire les graines proches du Mancala adverse limite leurs opportunités d'accumuler des points. Cela empêche l'adversaire de semer efficacement les graines pour remplir leur propre Mancala.

```
// Réduire les graines adverses proches de leur Mancala
int minimizeOpponentStones = 0;
for (int i = 7; i < 12; i++) {
    minimizeOpponentStones += pos.board[i];
}
```

Heuristique 3 : Empêcher les opportunités de capture adverses

Cette heuristique est intégrée dans deux endroits :

1. ***Lors de l'évaluation des positions :***

Cette pénalité réduit l'évaluation des positions qui exposent des cases vides de manière à permettre des captures adverses.

```
if (pos.board[i] > 0 && nextIndex >= 0 && nextIndex < 6 && pos.board[nextIndex] == 0) {
    // Vérifie si un coup rend une case vulnérable pour une capture
    int oppositeIndex = 12 - nextIndex;
    preventOpponentCapture -= pos.board[oppositeIndex]; // Réduction proportionnelle aux graines à capturer
}
```

2. ***Lors de la génération des coups :***

Cette pénalité est appliquée dans possibleMoves pour des coups qui laissent des opportunités de capture à l'adversaire.

Rôle : Empêcher les opportunités de capture protège les graines d'être capturées par l'adversaire. Cela rend le jeu plus défensif et stratégique.

```
// Penalty for leaving opportunities for the opponent
if (!player && newPos.board[12 - i] > 0) {
    eval -= 5.0f; // Pénalisation de base pour opportunités adverses
    eval -= newPos.board[12 - i]; // Réduction proportionnelle aux graines capturables
}
```

Règles classiques implémentées

1.Redistribution des graines dans les pits :

Les graines d'un pit sélectionné sont redistribuées une par une dans les pits suivants.

Implémentée dans **makeMove(Position p, boolean player, Move move).**

```
board[index] = 0; // Vide le pit sélectionné
int currentIndex = index;

// Sème les graines une par une dans les cases suivantes
while (seeds > 0) {
    currentIndex = (currentIndex + 1) % 14;

    // Saute la grande case (Mancala) de l'adversaire
    if ((player && currentIndex == MancalaPosition.PROGRAM_MANCALA) ||
            (!player && currentIndex == MancalaPosition.HUMAN_MANCALA)) {
        continue;
    }

    board[currentIndex]++;
    seeds--;
}
```

2.Captures :

Si la dernière graine tombe dans un pit vide du côté du joueur, les graines opposées sont capturées.

Implémentée dans **makeMove().**

```
// Capture : Si la dernière graine tombe dans un pit vide du côté du joueur
if (board[currentIndex] == 1 &&
        ((player && currentIndex >= 0 && currentIndex <= 5) ||
                (!player && currentIndex >= 7 && currentIndex <= 12))) {
    int oppositeIndex = 12 - currentIndex;
    int mancala = player ? MancalaPosition.HUMAN_MANCALA : MancalaPosition.PROGRAM_MANCALA;

    // Capture les graines opposées et la graine finale
    board[mancala] += board[oppositeIndex] + board[currentIndex];
    board[oppositeIndex] = 0;
    board[currentIndex] = 0;
}
```

3.Tours supplémentaires :

Un joueur rejoue si sa dernière graine tombe dans son Mancala.

Implémentée dans **makeMove().**

```
// Vérifie si le joueur gagne un tour supplémentaire
int ownMancala = player ? MancalaPosition.HUMAN_MANCALA : MancalaPosition.PROGRAM_MANCALA;
pos.extraTurn = (currentIndex == ownMancala);
```

4.principe de fin de jeu :

Le jeu se termine lorsqu'un côté du plateau (soit les pits du joueur humain, soit ceux du programme) est vide. Les graines restantes dans les pits du côté encore actif sont transférées dans le Mancala du joueur correspondant. Le joueur avec le plus de graines dans son Mancala remporte la partie.

Implémentée dans **drawnPosition(Position p).**

```java
// Si l'un des côtés est vide, transférer les graines restantes dans le Mancala correspondant
if (humanEmpty || programEmpty) {
    if (!humanEmpty) { // PROGRAM est vide, transfert les graines restantes de HUMAN
        for (int i = 0; i < 6; i++) {
            pos.board[MancalaPosition.HUMAN_MANCALA] += pos.board[i];
            pos.board[i] = 0;
        }
    } else { // HUMAN est vide, transfert les graines restantes de PROGRAM
        for (int i = 7; i < 13; i++) {
            pos.board[MancalaPosition.PROGRAM_MANCALA] += pos.board[i];
            pos.board[i] = 0;
        }
    }
    return true; // Le jeu est terminé
}

return false; // Le jeu continue
```

5. Détermine si un joueur a gagné :

Une fois le jeu terminé, les scores finaux sont dans les emplacements **Mancala** des deux joueurs.

```java
@Override  no usages
public boolean wonPosition(Position p, boolean player) {
    MancalaPosition pos = (MancalaPosition) p;

    // Vérifie si le jeu est terminé en utilisant drawnPosition
    if (!drawnPosition(p)) {
        return false; // Pas de gagnant tant que le jeu n'est pas terminé
    }

    // Si le jeu est terminé, compare les scores des Mancalas
    int humanScore = pos.board[MancalaPosition.HUMAN_MANCALA];
    int programScore = pos.board[MancalaPosition.PROGRAM_MANCALA];
    if (player) {
        return humanScore > programScore; // HUMAN gagne si son score est plus élevé
    } else {
        return programScore > humanScore; // PROGRAM gagne si son score est plus élevé
    }
}
```

Interface Utilisateur

1.Menu Principal :

Permet aux utilisateurs de choisir entre « Humain vs Humain » et « Humain vs Ordinateur ».

Code associé : MancalaMenuUI.

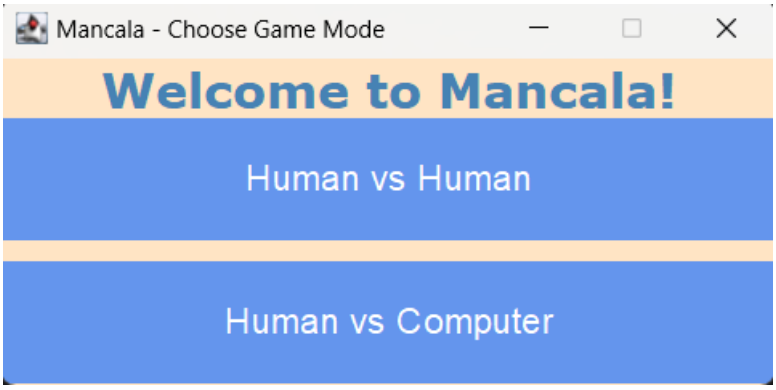

Figure 2 : interface du Menu Principal.

2. Écran du jeu principal :

Fournir une interface graphique pour jouer au jeu Mancala selon les règles classiques.

Code associé : MancalaGameUI et MancalaGUI.

a. Entre Humain vs Humain :

- Affiche les pits de chaque joueur et leurs scores respectifs.
- Indique clairement le tour actuel ("Playesr1's Turn", "Players2's Turn", etc.).
- Inclut des boutons de contrôle pour sauvegarder, charger ou quitter la partie.

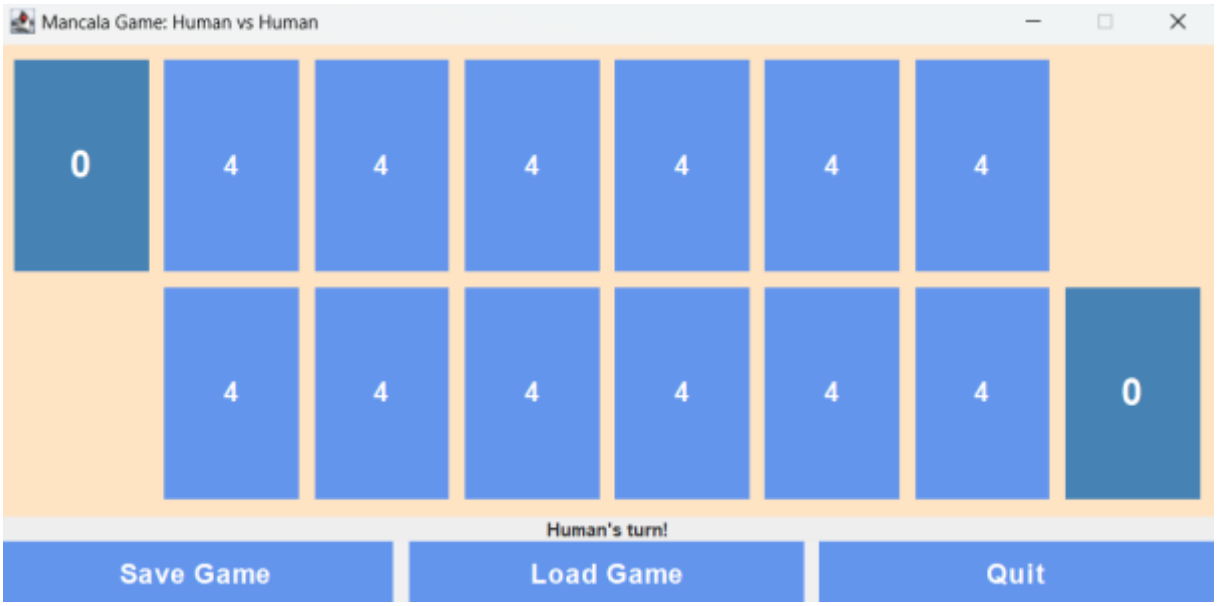

Figure 3 : interface du jeu en mode Human vs Human.

b. Entre Humain vs computer :

- Affiche les pits de chaque joueur et leurs scores respectifs.
- Indique clairement le tour actuel ("Playesr1's Turn", "Computer's Turn", etc.).
- Inclut des boutons de contrôle pour sauvegarder, charger, demander de l'aide ou quitter la partie.

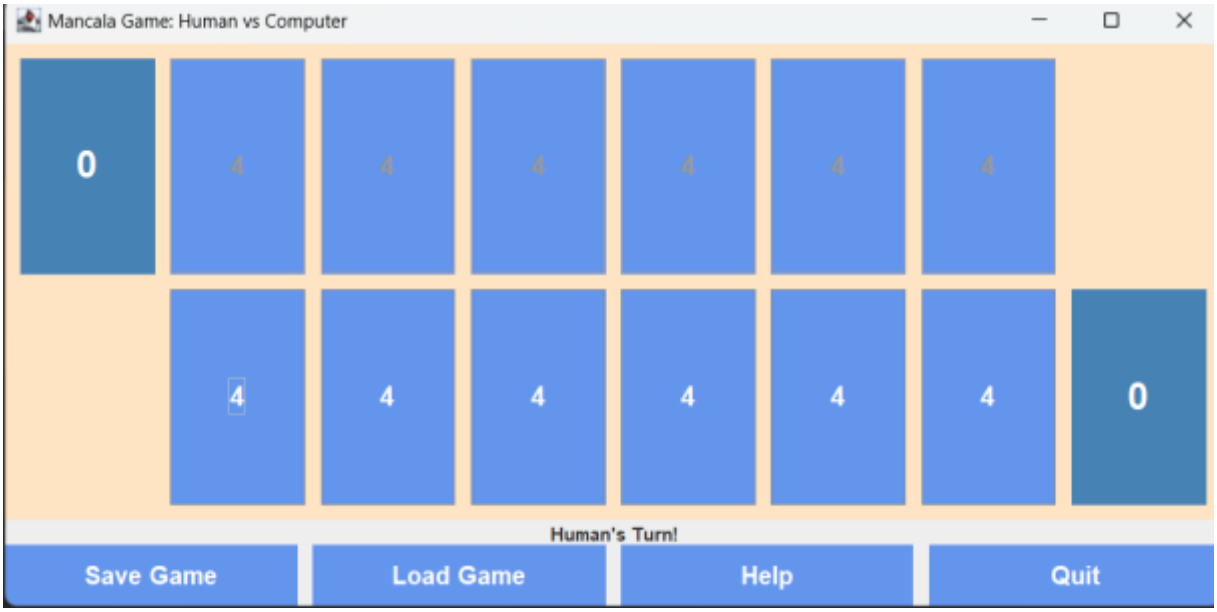

Figure 4 : interface du jeu en mode Human vs Computer.

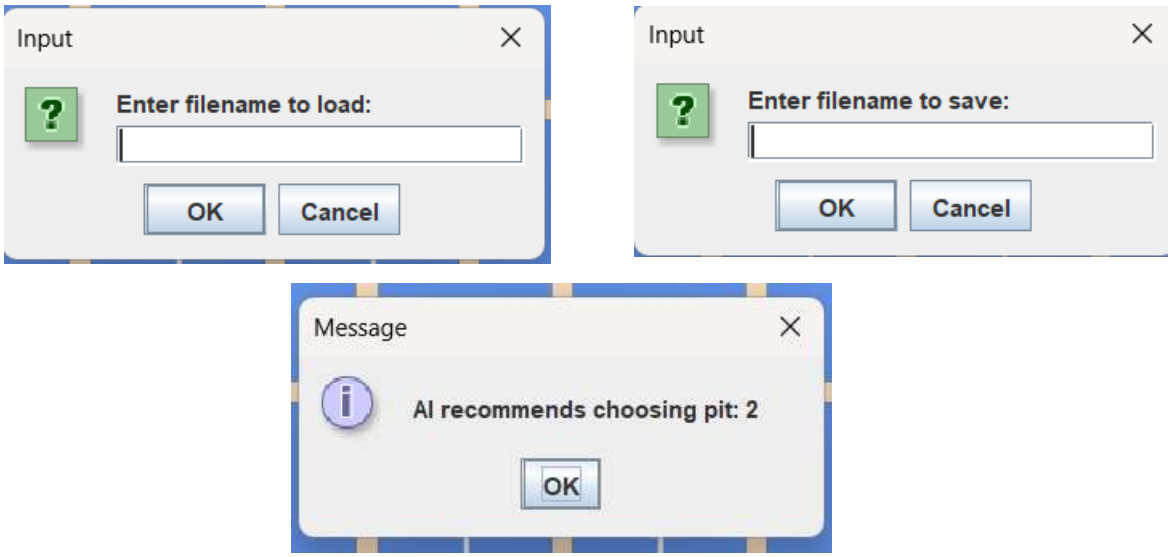

Figure 5 : interfaces contrôles pour sauvegarder, charger et demander de l'aide.

3.Écran de sélection de la difficulté :

Uniquement pour le mode Humain vs Ordinateur. Permettant au joueur de choisir entre trois niveaux de difficulté : **Simple**, **Medium**, et **Hard**.

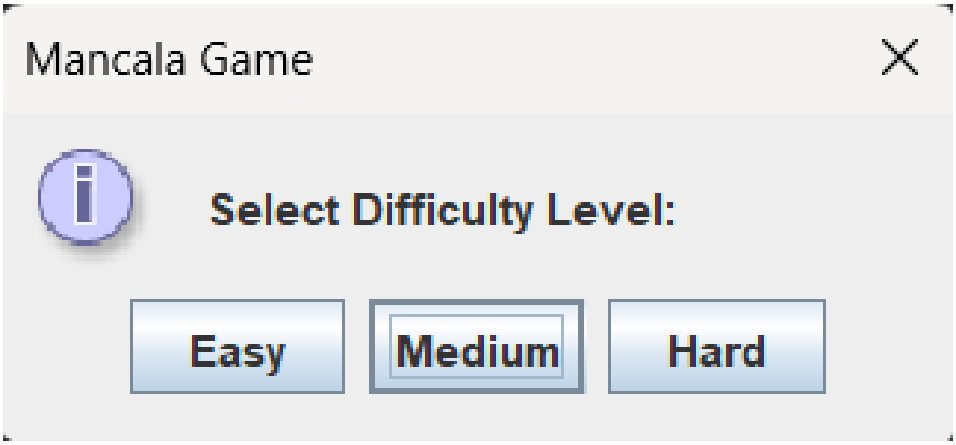

Figure 5 : interface de sélection de la difficulté.

Conclusion

Ce projet a permis de développer une application Java implémentant le jeu traditionnel du Mancala, tout en intégrant des algorithmes de recherche adversariale, tels que l'algorithme alpha-bêta, afin d'améliorer la stratégie de la machine. L'application permet non seulement de jouer contre un adversaire humain, mais aussi contre l'ordinateur, avec une fonctionnalité d'aide permettant de guider le joueur lors de ses parties. En outre, des heuristiques ont été utilisées pour évaluer les positions du jeu et orienter les décisions de la machine.

La prise en compte des contraintes du jeu, telles que la distribution des pierres et la gestion des captures, a permis de reproduire fidèlement les règles traditionnelles tout en optimisant l'expérience de jeu. Enfin, la possibilité de sauvegarder et reprendre une partie a ajouté une dimension supplémentaire de confort pour l'utilisateur. Ce projet illustre l'application des concepts d'intelligence artificielle dans le domaine des jeux de société et met en valeur l'importance de la stratégie dans les jeux adversariaux.