Université Côte d'Azur

Master 1 Informatique – Parcours Intelligence Artificielle

Projet de Game Programming

Développement d'une Intelligence Artificielle pour le Jeu d'Awalé (Variante Colorée)

Réalisé par :

Ikram BENCHALAL
Aya HADDOUN

Année universitaire 2025-2026

Table des matières

1 Introduction

Ce rapport présente le travail réalisé dans le cadre du projet de *Game Programming* du Master 1 Informatique, parcours Intelligence Artificielle, à l'Université Côte d'Azur.

L'objectif était de concevoir une intelligence artificielle capable de jouer à une variante complexe du jeu d'Awalé. Cette version se distingue du jeu classique par plusieurs aspects :

— Un plateau de **16 trous** (au lieu de 12).
— Trois types de graines : **Rouges (R)**, **Bleues (B)** et **Transparentes (T)**.
— Des règles de distribution spécifiques selon la couleur choisie.
— Une contrainte de temps stricte de **2 secondes** par coup.

Notre bot devait être capable de s'interfacer avec un arbitre Java fourni, de respecter scrupuleusement les règles du jeu, et de proposer une stratégie compétitive face à d'autres adversaires.

2 Architecture du Projet

Nous avons choisi de développer le projet en **Python** pour sa flexibilité et sa rapidité de prototypage, puis de le compiler en exécutable (`.exe`) via **PyInstaller** pour l'interfacer avec l'arbitre Java.

2.1 Structure des Fichiers

Le projet est organisé de manière modulaire :

Fichier	Description
`game.py`	Moteur du jeu : modélisation du plateau, règles de distribution et de capture, validation des coups.
`bot.py`	Intelligence artificielle : algorithme Minimax avec élagage Alpha-Beta et gestion du temps.
`player_adapter.py`	Interface de communication avec l'arbitre (lecture/écriture stdin/stdout).
`main.py`	Script de test local pour simuler des parties sans arbitre.
`Arbitre.java`	Arbitre Java fourni, modifié pour lancer notre exécutable.

2.2 Description Détaillée des Modules

2.2.1 game.py – Le Moteur du Jeu

Ce module implémente la classe `AwaleGame` qui encapsule toute la logique du jeu :

— **Représentation du plateau** : Un tableau de 16 cases, chaque case contenant un triplet `[R, B, T]` représentant le nombre de graines de chaque couleur.
— **Gestion des joueurs** : Le Joueur 1 possède les trous impairs (1, 3, 5...), le Joueur 2 les trous pairs (2, 4, 6...).
— **Validation des coups** : Vérification stricte de la légalité d'un coup avant son exécution.
— **Distribution des graines** : Implémentation des règles spécifiques :
  — **R** : Distribution dans tous les trous.
  — **B** : Distribution uniquement dans les trous adverses.

— **TR/TB** : Prise des transparentes ET des colorées correspondantes, distribution des transparentes en premier.
— **Capture** : Analyse en chaîne à partir du dernier trou semé, capture si le total de graines est 2 ou 3.

2.2.2 bot.py – L'Intelligence Artificielle

C'est le cœur stratégique du projet. Ce module implémente la classe `MinimaxBot` détaillée dans la section suivante.

2.2.3 player_adapter.py – L'Interface de Communication

Ce script fait le lien entre notre bot Python et l'arbitre Java :

1. Lecture des commandes sur l'entrée standard (`START` ou coup adverse).
2. Mise à jour de l'état local du jeu.
3. Interrogation du bot pour obtenir le meilleur coup.
4. Validation de sécurité (`pick_safe_move`) pour éviter les coups invalides.
5. Envoi du coup sur la sortie standard.

3 Choix d'Implémentation

3.1 Algorithme Minimax avec Élagage Alpha-Beta

Nous avons opté pour l'algorithme **Minimax**, un classique des jeux à deux joueurs à somme nulle. Le principe est simple :
— À notre tour (MAX), on cherche à maximiser notre score.
— Au tour de l'adversaire (MIN), on suppose qu'il jouera pour minimiser notre gain.
Pour optimiser les performances, nous avons implémenté l'**élagage Alpha-Beta**. Cette technique permet de couper les branches de l'arbre de recherche qui ne peuvent pas influencer la décision finale :

```
alpha = max(alpha, eval)
if beta <= alpha:
    break  # Coupure Beta
```

Listing 1 – Élagage Alpha-Beta dans minimax()

En pratique, cela réduit considérablement le nombre de nœuds explorés, permettant d'atteindre des profondeurs plus importantes dans le temps imparti.

3.2 Iterative Deepening (Approfondissement Itératif)

La contrainte de 2 secondes par coup nous a conduits à implémenter l'**Iterative Deepening**. Le bot explore l'arbre de jeu en augmentant progressivement la profondeur :

1. Recherche à profondeur 1 → sauvegarde du meilleur coup.
2. Recherche à profondeur 2 → mise à jour si meilleur coup trouvé.
3. Recherche à profondeur 3, 4, 5...
4. **Interruption** dès que le temps limite (1.9s avec marge de sécurité) est atteint.

Cette approche garantit que le bot renvoie **toujours** un coup valide, même si le temps est court, tout en explorant le plus profondément possible quand le temps le permet.

```
for d in range(1, max_depth_to_search + 1):
    if self._time_up(start_time):
        break
    try:
        move, score = self.minimax_root(game, d, start_time)
        if move is not None:
            best_move = move
    except TimeoutError:
        break
return best_move
```

Listing 2 – Iterative Deepening dans get_best_move()

3.3 Gestion Stricte du Temps

Pour éviter les disqualifications par timeout, nous avons utilisé `time.perf_counter()` (plus précis que `time.time()`) et fixé une limite interne de **1.9 secondes** (marge de 100ms pour la communication I/O).

4 Fonction d'Évaluation

La fonction d'évaluation est **cruciale** car elle guide les décisions du bot. Elle attribue un score à chaque état du jeu, permettant au Minimax de comparer les différentes branches.

4.1 Composantes de la Fonction

Notre fonction d'évaluation prend en compte plusieurs facteurs pondérés :

4.1.1 1. Différence de Score (Poids : $\times 10$)

C'est le critère le plus important. Le but du jeu étant d'atteindre 49 graines capturées, la différence de score est primordiale :

$$\text{score\_diff} = (\text{mon\_score} - \text{score\_adversaire}) \times 10$$

4.1.2 2. Détection de Victoire/Défaite Immédiate

Si un joueur a dépassé 48 graines, la partie est gagnée/perdue. On retourne une valeur extrême :

```
if my_score > 48:
    return 10000   # Victoire certaine
if opp_score > 48:
    return -10000  # Defaite certaine
```

4.1.3 3. Potentiel de Capture (Poids : +3 par trou)

Nous analysons les trous adverses contenant 1 ou 2 graines. Ces trous sont facilement capturables au prochain tour :

$$\text{capture\_potential} = \sum_{\text{trou adverse}} \begin{cases} 3 & \text{si total} \in \{1, 2\} \\ 0 & \text{sinon} \end{cases}$$

4.1.4 4. Mobilité (Poids : ×2, Pénalité si ≤ 2)

La mobilité représente le nombre de coups possibles. Un joueur avec peu de mobilité risque le blocage :

```
mobility_bonus = my_mobility * 2
if my_mobility <= 2:
    mobility_bonus -= 10  # Penalite de risque de blocage
```

4.1.5 5. Contrôle des Graines

Avoir plus de graines dans ses propres trous que l'adversaire indique un meilleur contrôle du jeu :

$$\text{seed\_control} = \frac{\text{mes\_graines} - \text{graines\_adverses}}{2}$$

4.2 Formule Finale

La fonction d'évaluation combine tous ces éléments :

$$\boxed{\text{Eval} = \underbrace{(\Delta\text{score}) \times 10}_{\text{priorité max}} + \text{capture\_potential} + \text{mobility\_bonus} + \text{seed\_control}}$$

```
def evaluate(self, game):
    my_score = game.scores[self.player_id]
    opp_score = game.scores[1 - self.player_id]

    # Victoire ou defaite immediate
    if my_score > 48:
        return 10000
    if opp_score > 48:
        return -10000

    # Indices de mes trous et ceux de l'adversaire
    my_start = 0 if self.player_id == 0 else 1
    opp_start = 1 - my_start

    # Compter mes graines et ma mobilite
    my_seeds = 0
    my_mobility = 0
    for i in range(my_start, 16, 2):
        r, b, t = game.board[i]
        my_seeds += r + b + t
        if r > 0: my_mobility += 1
```

```
        if b > 0: my_mobility += 1
        if t > 0: my_mobility += 2

    # Compter les graines adverses et le potentiel de capture
    opp_seeds = 0
    capture_potential = 0
    for i in range(opp_start, 16, 2):
        r, b, t = game.board[i]
        total = r + b + t
        opp_seeds += total
        if total == 1 or total == 2:
            capture_potential += 3

    # Bonus/Penalite de mobilite
    mobility_bonus = my_mobility * 2
    if my_mobility <= 2:
        mobility_bonus -= 10

    # Controle des graines
    seed_control = (my_seeds - opp_seeds) // 2

    # Formule finale
    return (my_score - opp_score) * 10 + capture_potential +
mobility_bonus + seed_control
```

Listing 3 – Fonction d'évaluation complète

4.3 Justification des Choix

- **Poids élevé sur le score** : Le score est l'objectif final du jeu. Un avantage de 5 points de score vaut plus que tous les autres bonus combinés.
- **Potentiel de capture** : Anticiper les captures futures permet d'orienter le jeu vers des positions avantageuses.
- **Mobilité** : Un joueur bloqué perd la partie. Cette heuristique évite les situations de blocage.
- **Graines transparentes comptées double pour la mobilité** : Elles offrent deux options de jeu (TR ou TB), d'où le bonus.

5 Difficultés Rencontrées et Solutions

5.1 Problème 1 : Coups Invalides (Désynchronisation)

Symptôme : Le bot était disqualifié pour "coup invalide" car il tentait de jouer des graines Rouges dans un trou où l'arbitre ne voyait que des Transparentes.

Cause : Notre simulation locale de la distribution des graines différait légèrement de celle de l'arbitre (ordre de distribution des transparentes).

Solution :

1. Correction de `game.py` pour respecter exactement les règles de l'arbitre.
2. Ajout d'une fonction `pick_safe_move()` qui vérifie la validité réelle d'un coup avant de l'envoyer et privilégie les coups "sûrs" (R ou B) en cas de doute.

5.2 Problème 2 : Timeout

Symptôme : Disqualification par dépassement des 2 secondes.
Solution :
— Passage de `time.time()` à `time.perf_counter()` pour une mesure plus précise.
— Limite interne fixée à 1.9s (marge de sécurité de 100ms).
— Profondeur maximale limitée à 5 pour éviter les explorations trop longues.
— Vérification du temps à chaque nœud exploré.

5.3 Problème 3 : Déploiement

Symptôme : L'arbitre Java ne pouvait pas exécuter directement nos scripts Python.
Solution : Compilation en exécutable autonome avec **PyInstaller** :

```
pyinstaller --onefile --name player player_adapter.py
```

6 Tests et Résultats

6.1 Tests Locaux

Nous avons validé le comportement du bot via des simulations locales intensives (Bot vs Bot) en utilisant `main.py`. Ces tests ont permis de :
— Vérifier l'absence de bugs sur des parties longues (jusqu'à 400 coups).
— Valider la fonction d'évaluation.
— Tester la robustesse face aux cas limites.

FIGURE 1 – Simulation d'une partie en local : validation de la logique de jeu et de l'IA.

6.2 Match contre un Adversaire Réel

Le test final a été effectué contre le bot de notre collègue **Kacem**. Le résultat est le suivant :

Joueur	Score	Résultat
Ikram Bot (Nous)	43	Défaite
Kacem Bot	44	Victoire

Bien que nous ayons perdu, l'écart de score est **extrêmement serré** (1 seul point de différence !), ce qui témoigne de la compétitivité de notre IA. Plus important encore, **aucune disqualification** n'a eu lieu : tous les coups étaient légaux et dans le temps imparti.

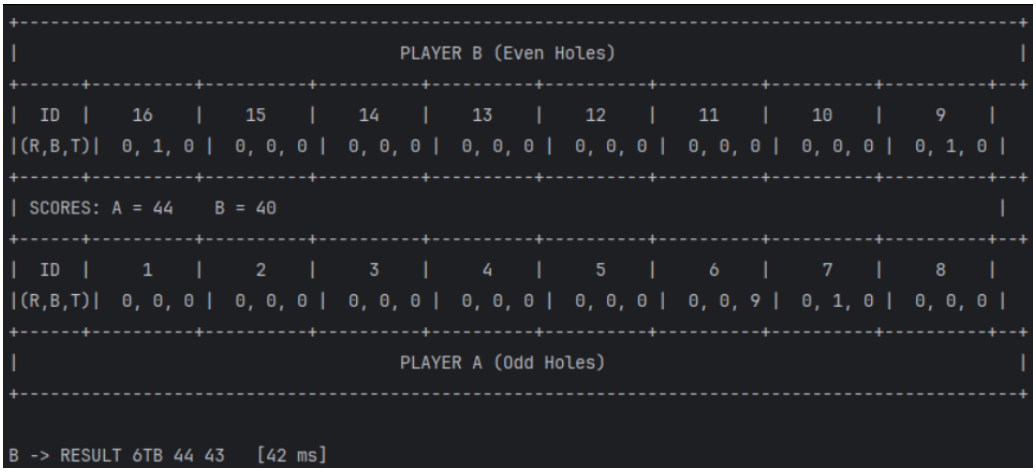

FIGURE 2 – Résultat du match contre Kacem : score serré et aucune erreur technique.

7 Conclusion et Perspectives

Ce projet nous a permis de mettre en œuvre une intelligence artificielle complète pour un jeu de stratégie combinatoire. Les principaux accomplissements sont :

- **Architecture modulaire** : Séparation claire entre logique de jeu, IA et communication.
- **Algorithme robuste** : Minimax + Alpha-Beta + Iterative Deepening.
- **Fonction d'évaluation multi-critères** : Score, mobilité, potentiel de capture, contrôle.
- **Gestion stricte du temps** : Aucun timeout lors des matchs finaux.
- **Fiabilité technique** : Plus de coups invalides après correction.

7.1 Améliorations Possibles

Pour aller plus loin, nous pourrions envisager :

- **Optimisation de la fonction d'évaluation** : Ajuster les poids via du *machine learning* ou des tests empiriques.
- **Table de transposition** : Mémoriser les états déjà évalués pour éviter les calculs redondants.
- **Opening book** : Pré-calculer les meilleurs coups d'ouverture.
- **Monte Carlo Tree Search (MCTS)** : Alternative au Minimax, potentiellement plus efficace pour les jeux à grand facteur de branchement.