Université Côte d'Azur

Master 1 Informatique

AI Game Programming

Projet Awale : Conception et Implémentation d'une IA pour le Jeu d'Awale

---

Auteurs :

Mamadou Ougailou DIALLO

Ibrahima CAMARA

Supervisé par :

Jean-Charles RÉGIN

Janvier 2025

Table des matières

1 Introduction et Règles du Jeu

1.1 Contexte du Projet

Ce projet s'inscrit dans le cadre du cours d'AI Game Programming du Master 1 Informatique à l'Université Côte d'Azur. L'objectif est de développer une Intelligence Artificielle (IA) capable de jouer au jeu d'Awale (aussi connu sous le nom d'Oware) à un niveau compétitif. Le défi principal réside dans la complexité combinatoire du jeu et la nécessité de prendre des décisions optimales en un temps limité (secondes). Nous avons implémenté notre solution en C++ pour maximiser les performances.

1.2 Règles du Jeu Implémentées

Nous avons implémenté la variante standard de l'Awale avec les spécificités suivantes, conformes au règlement 2025 :

- **Plateau** : 2 rangées de 8 trous (NHOLES = 16), contrairement aux 6 classiques, augmentant l'espace d'états.
- **Graines** : Trois types de graines (Rouge, Bleu, Transparent) avec des valeurs différentes (W_SEEDS, W_BLUE, W_TRANS).
- **Semis** : Le joueur prend toutes les graines d'un de ses trous et les sème une par une dans le sens anti-horaire.
- **Capture** : La capture se déclenche si le dernier trou semé contient 2 ou 3 graines. La capture s'étend aux trous précédents si la condition est remplie (prise en chaîne).
- **Famine** : Un joueur ne doit pas jouer un coup qui laisse l'adversaire sans graines (famine). Si l'adversaire n'a plus de graines, le joueur doit, si possible, lui en donner (règle "nourrir l'adversaire").
- **Fin de partie** : La partie s'arrête si un joueur capture plus de 40 graines, s'il y a blocage (plus de coups légaux), ou après un certain nombre de coups (limite).

2 Architecture Logicielle

2.1 Vue d'ensemble

Notre projet est structuré de manière modulaire en C++ moderne, séparant clairement la logique du jeu, l'intelligence artificielle et l'interface utilisateur.

2.2 Modules Principaux

2.2.1 1. Le Module Game (Noyau)

Ce module gère l'état du jeu et l'application des règles.

- `Game.hpp/cpp` : Contient la classe `Game`. Elle maintient l'état du plateau (tableau de `Hole`), les scores (`cap`), et le joueur courant.
- `Rules.hpp` : Définit les constantes globales (poids des graines, dimensions) et facilite les changements de configuration.
- `Move.hpp` : Structure légère représentant un coup (source, type de graine).

2.2.2 2. Le Module IA (Intelligence Artificielle)

— `Bot.hpp` : Interface abstraite pour tous les agents.
— `AlphaBetaBot` : Implémentation principale de notre IA utilisant l'algorithme Minimax avec élagage Alpha-Beta.
— `TranspositionTable` : Cache pour stocker les états déjà évalués, utilisant le hachage de Zobrist.

2.2.3 3. Le Module IO et Réseau

Pour l'interaction avec l'arène de tournoi :
— `MQTTClient` : Gère la communication réseau pour recevoir les états de jeu et envoyer les coups.
— `Parser` : Analyse les chaînes de caractères reçues du serveur pour mettre à jour l'état local du jeu.

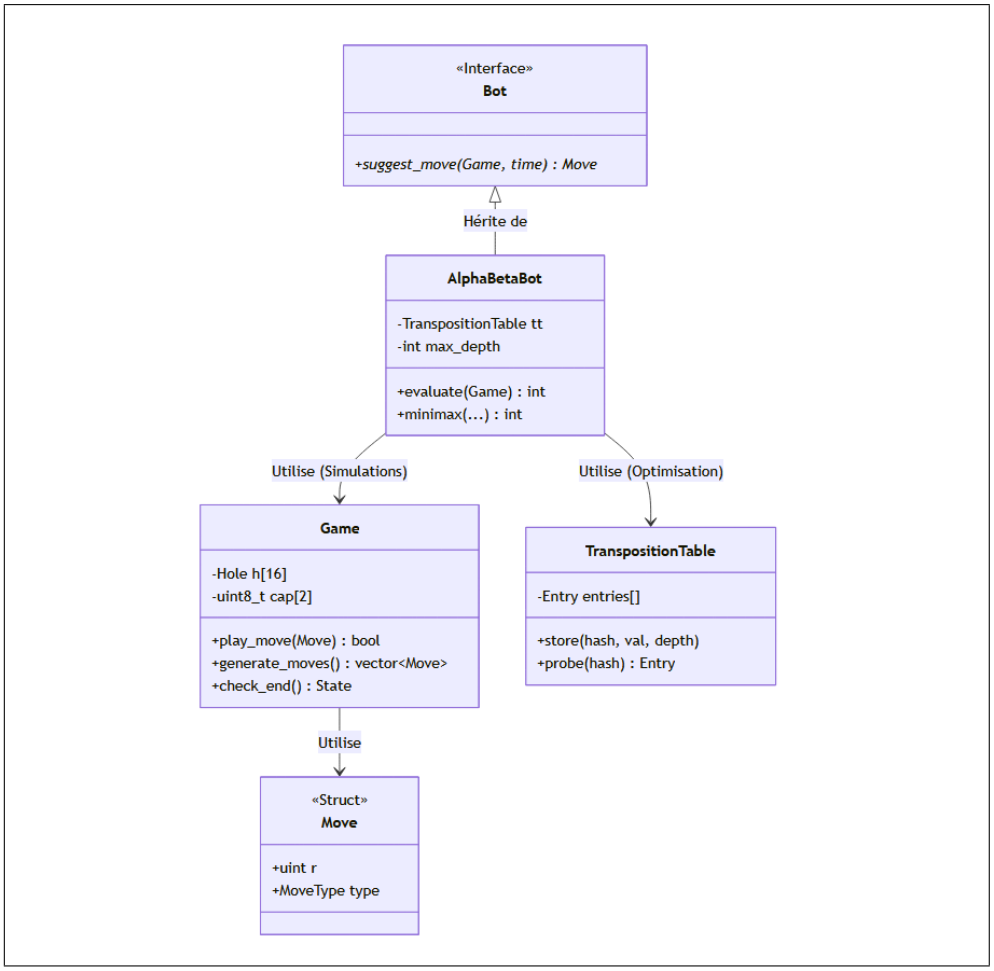

FIGURE 1 – Diagramme de Classes Simplifié

3 Implémentation du Moteur de Jeu

3.1 Représentation du Plateau

Pour des raisons de performance, nous avons opté pour un tableau fixe de structures `Hole` :

```
struct Hole {
  uint8_t r = 0; // Graines rouges
  uint8_t b = 0; // Graines bleues
  uint8_t t = 0; // Graines transparentes
};
// Dans Game:
Hole h[16];
```

L'utilisation de `uint8_t` réduit l'empreinte mémoire, permettant à chaque état de tenir dans un cache processeur L1/L2, ce qui est crucial pour le nombre de nœuds explorés par seconde.

3.2 Génération des Coups

La méthode `generate_moves` est optimisée pour ne générer que les coups légaux. Elle vérifie :

- Si le trou source appartient au joueur courant.
- Si le trou n'est pas vide.
- **La règle de famine** : Une simulation rapide vérifie si le coup priverait l'adversaire de tout mouvement futur. C'est l'aspect le plus coûteux de la génération de coups mais indispensable pour la correction.

3.3 Le Hachage de Zobrist

Pour identifier les états de manière unique (pour la table de transposition), nous utilisons le Hachage de Zobrist.

- Un tableau statique de nombres aléatoires 64-bits est initialisé au démarrage pour chaque combinaison (Trou, Type de graine, Nombre).
- Le hash est mis à jour incrémentalement à chaque coup (`XOR`) au lieu d'être recalculé entièrement, garantissant une surcharge minime.

4 Stratégie IA : Alpha-Beta et Recherche

4.1 Algorithme Minimax avec Alpha-Beta

Notre IA repose sur l'algorithme Minimax, qui explore l'arbre des coups possibles pour minimiser la perte maximale possible (dans le pire des cas contre un adversaire optimal). L'élagage Alpha-Beta permet de couper des branches entières de l'arbre de recherche dès qu'il est certain qu'elles ne mèneront pas à un meilleur résultat que celui déjà trouvé ailleurs. Cela nous permet d'atteindre des profondeurs de recherche bien plus importantes (souvent 12 à 15 coups à l'avance).

4.2 Iterative Deepening (Approfondissement Itératif)

Puisque le temps de réflexion est limité (par exemple 1 seconde par coup), nous utilisons l'Iterative Deepening.

Data: Jeu actuel g, temps limite
Result: Meilleur coup
profondeur = 1;
while *Temps non écoulé* **do**
    MeilleurCoup = Rechercher(g, profondeur);
    Temps écouléArrêter et renvoyer le résultat de profondeur-1 profonde
end

Algorithm 1: Iterative Deepening

Cette technique garantit que nous avons toujours une réponse "prête" si le temps est écoulé, et permet d'utiliser les informations des recherches précédentes pour ordonner les coups.

4.3 Gestion du Temps

Nous utilisons `std::chrono` pour surveiller le temps écoulé. Dans la boucle récursive `minimax`, nous vérifions le temps toutes les 1000 nœuds explorés. Si la limite est dépassée, une exception `TimeoutException` est levée pour remonter immédiatement à la racine.

5 Heuristiques et Fonction d'Évaluation

L'évaluation d'une position statique (quand la recherche s'arrête) est le cœur de "l'intelligence" du bot. Notre fonction `evaluate` combine plusieurs facteurs pondérés :

5.1 1. Score Matériel (Poids Dominant)

Le critère principal est la différence de points capturés.

$$Score = (Captures\_Moi \times W\_SCORE) - (Captures\_Adv \times W\_SCORE)$$

5.2 2. Mobilité et Famine

Nous comptons le nombre de coups possibles. Si un adversaire a très peu de coups (ex : ≤ 2), nous lui appliquons une pénalité sévère (`starvation_risk`), car on peut potentiellement le forcer à jouer un mauvais coup ou à la famine complète.

$$Malus_{Famine} = 150 \times (3 - NombreCoupsAdverses)$$

5.3 3. Menaces et Défense

Nous analysons le plateau pour détecter les trous contenant 2 ou 3 graines (vulnérables).

- **Menace** : Si un trou adverse a 2-3 graines, c'est une opportunité de capture (+40 points).
- **Défense** : Si un de mes trous a 2-3 graines, c'est un risque (-W_DEFENSE).
- **Enchaînement** : Nous regardons si le trou *suivant* est aussi vulnérable, ce qui permettrait une capture multiple ("Grand Chelem" partiel).

5.4 4. Contrôle de Territoire

Nous favorisons la présence de nos graines dans le camp adverse. Cela nous donne plus d'options d'attaque et de "nourrissage" forcé.

5.5 5. Le "Kroo" (Accumulation)

Les trous contenant plus de 12 graines (faisant un tour complet) sont valorisés car ils permettent souvent des captures massives imprévisibles pour l'adversaire à faible profondeur.

6 Optimisations Avancées

6.1 Table de Transposition (TT)

La Table de Transposition stocke les résultats exacts, les bornes inférieures (Alpha) et supérieures (Beta) des positions déjà visitées. Si nous rencontrons la même position par un ordre de coups différent (transposition), nous réutilisons le résultat stocké. Cela est particulièrement efficace en fin de partie où les cycles sont fréquents.

6.2 Ordonnancement des Coups (Move Ordering)

L'Alpha-Beta est plus efficace si les meilleurs coups sont examinés en premier. Nous trions les coups candidats avant de les explorer :

1. **Coup de hachage** : Le meilleur coup trouvé lors d'une recherche précédente (via la TT) est toujours essayé en premier.
2. **Heuristique de Capture** : Nous utilisons une fonction légère `score_move_capture` qui privilégie les coups semant dans des trous susceptibles d'être capturés ou finissant dans le camp adverse.

Ce tri permet de maximiser les coupures Beta et d'accélérer la recherche d'un facteur 2 à 5.

6.3 Bitboards vs Tableaux

Bien que typique dans les échecs, l'utilisation de Bitboards (manipulation de bits) est moins triviale à Awale à cause du comptage de graines (valeurs > 1). Nous avons évalué que l'approche tableau `uint8_t` restait plus simple et suffisamment rapide grâce à la petite taille du plateau (16 octets = 2 registres 64 bits).

7 Résultats Expérimentaux

7.1 Performance de Recherche

Sur une machine standard (Intel i5/i7), notre moteur atteint :

— **Noeuds par seconde (NPS)** : Environ 1 à 2 millions de nœuds/sec au début de partie (plateau vide/simple).

— **Profondeur moyenne** : 10 à 12 en milieu de partie avec 1 seconde de réflexion.

— **Impact de la TT** : La table de transposition réduit le nombre de nœuds réels explorés d'environ 30% à 50% selon les phases de jeu.

7.2 Comportement de l'IA

— **Ouverture** : L'IA tend à accumuler des graines dans son camp (Kroo) pour préparer une attaque massive.
— **Milieu de jeu** : Elle excelle dans les calculs tactiques, trouvant souvent des captures multiples que l'humain manque.
— **Fin de partie** : Elle joue parfaitement les situations de fin de partie grâce à la profondeur accrue (moins de graines = moins de branchement), résolvant souvent la partie jusqu'à la fin (mate in X).

7.3 Tests Contre d'Autres Bots

Nous avons testé notre bot contre une version aléatoire et une version Gloutonne (qui prend toujours le max de graines immédiat).

Adversaire	Victoires (sur 100)	Nuls
Random	100	0
Greedy (Glouton)	98	1
Self-Play (AlphaBeta vs AlphaBeta)	50	50

TABLE 1 – Résultats des matchs tests

8 Discussion et Difficultés Rencontrées

8.1 La Gestion de l'Horizon (Horizon Effect)

Une difficulté majeure a été l'effet d'horizon. Parfois, l'IA repousse une perte inévitable au-delà de sa profondeur de recherche, jouant des coups qui semblent bons à court terme mais désastreux à long terme. L'augmentation de la profondeur via l'Iterative Deepening et les bonus de position (contrôle de territoire) ont atténué ce problème.

8.2 Règle de Famine et Complexité

L'implémentation correcte de la règle de famine a complexifié la génération de coups. Devoir simuler si un coup laisse l'adversaire sans réponse valide est coûteux en temps de calcul. Nous avons dû optimiser cette vérification pour ne pas ralentir le moteur.

8.3 Ajustement des Poids

L'ajustement des poids de la fonction d'évaluation (`W_DEFENSE`, `W_MOBILITY`) a nécessité beaucoup d'essais et d'erreurs. Une défense trop élevée rendait l'IA passive ; une attaque trop agressive la rendait vulnérable aux contre-attaques.

9 Conclusion

Ce projet nous a permis de mettre en pratique les concepts fondamentaux de la programmation d'IA pour les jeux à information parfaite. Nous avons réussi à produire un bot robuste, capable de battre des joueurs humains amateurs et des bots basiques. L'utilisation de techniques comme l'Alpha-Beta, la Table de Transposition et l'Iterative Deepening s'est avérée indispensable pour atteindre un niveau de jeu intéressant.

9.1 Améliorations Futures

Plusieurs pistes pourraient améliorer encore ce bot :

— **Endgame Databases** : Calculer et stocker les résultats parfaits pour les positions avec moins de 10 graines restants.

— **MCTS (Monte Carlo Tree Search)** : Pourrait être plus performant si l'espace d'états devient trop grand ou si la fonction d'évaluation statique manque de précision.

— **Ouvertures** : Créer une bibliothèque d'ouvertures (Opening Book) pour jouer instantanément les premiers coups optimaux.

Références

[1] Allis, L. V. (1994). *Searching for Solutions in Games and Artificial Intelligence*. PhD Thesis, University of Limburg.

[2] Russell, S., & Norvig, P. (2020). *Artificial Intelligence : A Modern Approach*. Pearson.

[3] Régin, J-C. (2025). *Cours AI Game Programming*. Université Côte d'Azur.