Implémentation et optimisation d'un modèle d'intelligence artificielle pour le jeu Reversi

GHODBANE Rachid - LAICHE Khayr Eddine - BAADACHE Nizar

Licence 2 Informatique - Université Jean Monnet 2024/2025

Résumé

Ce rapport présente le travail réalisé dans le cadre d'un projet de programmation impérative de deuxième année de licence informatique, ayant pour but de concevoir une Intelligence Artificielle (IA) capable de jouer au jeu Reversi. Nous détaillons notre approche incrémentale, les structures de données utilisées, la construction de l'arbre de jeu, les algorithmes de recherche (Minimax, Alpha-Bêta, NegaScout), ainsi que les optimisations mémoire et les améliorations heuristiques apportées. Les performances sont évaluées expérimentalement sur plusieurs phases de jeu.

Code source du projet : https://github.com/rvsh0x/AI_Reversi.

Table des matières

1 Approche de l'IA

Notre développement s'est fait de manière incrémentale, en suivant une progression en six niveaux d'IA, du comportement le plus simple (joueur aléatoire) jusqu'aux optimisations avancées.

1.1 Développement incrémental

Nous avons mis en place :

— 6 niveaux d'IA, du plus simple (aléatoire) jusqu'aux versions optimisées.

— Une intégration progressive des fonctionnalités pour isoler les effets de chaque amélioration.

1.2 Évolution de la fonction d'évaluation

La qualité du jeu dépend fortement de la fonction d'évaluation. Nous l'avons enrichie progressivement :

— Nombre de pions (différence IA/adversaire).

— Détection de séries de 2 puis 3 pions alignés.

— Pondération des régions stratégiques.

— Critères avancés : mobilité, stabilité, parité, etc.

1.3 Algorithmes de recherche

Les algorithmes de recherche ont été intégrés par étapes :

— Minimax.

— Alpha-Bêta.

— NegaScout avec variantes de tri des coups.

1.4 Optimisation mémoire

Pour maîtriser l'explosion combinatoire, nous avons adopté :

— Un seul plateau en mémoire.

— Un historique des coups permettant l'annulation sans duplication de plateau.

1.5 Validation expérimentale

Nous avons mené des tests sur :

— le temps d'exécution

— le nombre de nœuds explorés

— la mémoire consommée

sur plusieurs plateaux correspondant à différentes phases de jeu.

1.6 Itération et décisions

Chaque amélioration a été soumise au cycle :

$$\text{mesure} \rightarrow \text{comparaison} \rightarrow \text{intégration si gain significatif.}$$

2 Structures de données

2.1 Plateau

Le plateau de Reversi est représenté par un tableau 2D $[8 \times 8]$ stockant l'état du jeu. Une énumération `Joueur` distingue `VIDE`, `NOIR` et `BLANC`. Cette structure est la base de l'IA : chaque position de l'arbre utilise le même modèle logique.

2.2 Nœud et arbre de jeu

Chaque nœud contient :

- un `Plateau`
- la position du coup joué
- des pointeurs vers ses enfants
- le nombre d'enfants
- une évaluation heuristique

Elle permet l'exploration via Minimax, Alpha-Bêta et NegaScout.

2.3 HistoriqueCoup

Pour éviter de dupliquer les plateaux :

- un seul `Plateau` est conservé
- l'historique enregistre pour chaque coup :
  - la case jouée
  - la liste des pions retournés

Ainsi, annuler un coup ne nécessite pas de recopier l'intégralité du plateau.

3 Arbre de jeu, Minimax et Alpha-Bêta

3.1 Définition de l'arbre de jeu

L'arbre de jeu représente tous les états possibles après une série de coups :

— La racine est l'état actuel du plateau.

— Les enfants sont les coups possibles de l'ordinateur.

— Les petits-enfants sont les réponses possibles du joueur, etc.

3.2 Construction récursive

1. Créer le nœud racine avec l'état actuel du plateau.
2. Identifier tous les coups légaux pour le joueur courant.
3. Pour chaque coup : simuler le nouveau plateau.
4. Créer un nœud enfant associé.
5. Continuer récursivement avec le joueur adverse.
6. Stopper à la profondeur souhaitée.

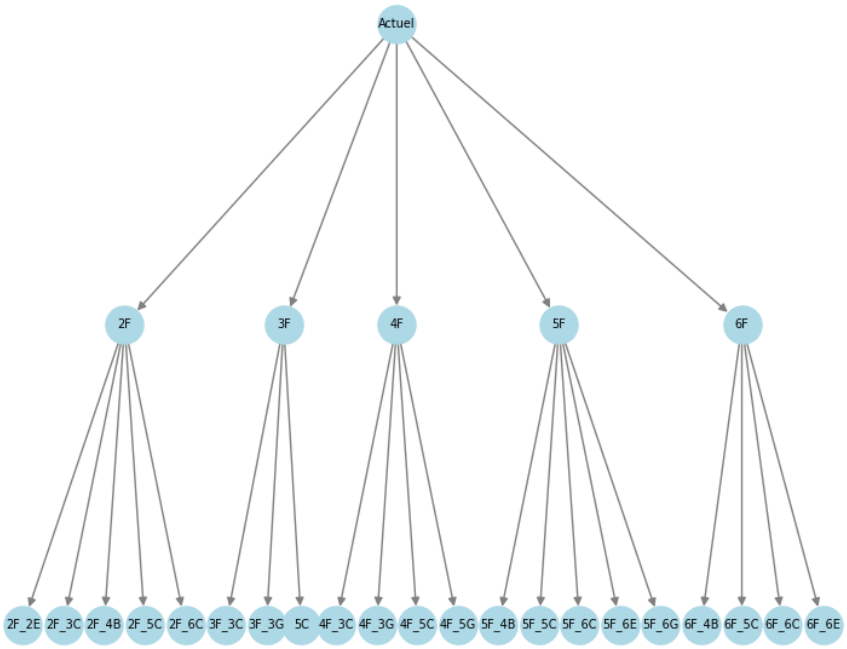

FIGURE 1 – Représentation simplifiée d'un arbre de jeu de profondeur 2 pour Reversi.

3.3 Évolution de la fonction d'évaluation

Rôle. La fonction d'évaluation assigne une valeur numérique à un état de plateau. Elle guide la prise de décision de Minimax/Alpha-Bêta : plus elle est précise, plus l'IA joue intelligemment.

Étape 3 : évaluation basique. Différence de pions entre IA et adversaire :

$$\text{score} = \sum_{cases} \begin{cases} +1 & \text{si pion IA} \\ -1 & \text{si pion adverse} \\ 0 & \text{sinon} \end{cases}$$

Étape 4 : ensembles consécutifs. Conservation de l'évaluation basique et ajout de :

- détection de séries alignées (2 et 3+),
- marquage pour éviter les comptages multiples.

Étape 5 : évaluation stratégique complète. Positions valorisées selon leur importance :

- Coins : +15 (très stables),
- Cases adjacentes aux coins : −7 (dangereuses)
- Bords : +5
- Séquences en bord : +10

avec adaptation selon la phase de jeu.

3.4 Algorithme Minimax

Algorithm 1 Algorithme Minimax

Require: Arbre a, profondeur d, booléen $estMax$
Ensure: Valeur d'évaluation optimale

if $a.nbEnfants = 0$ **or** $d = 0$ **then**
  return $evaluationPlateau(a.plateau)$
end if
if $estMax$ **then**
  $best \leftarrow -10^6$
  for $i = 0$ à $a.nbEnfants - 1$ **do**
    $val \leftarrow minimax(a.enfants[i], d - 1, false)$
    $best \leftarrow \max(best, val)$
  end for
else
  $best \leftarrow 10^6$
  for $i = 0$ à $a.nbEnfants - 1$ **do**
    $val \leftarrow minimax(a.enfants[i], d - 1, true)$
    $best \leftarrow \min(best, val)$
  end for
end if
return $best$

3.5 Algorithme Alpha-Bêta

Principe général. Alpha-Bêta est une variante de Minimax qui élaguie des branches inutiles. Elle maintient deux bornes dynamiques :

— α : meilleure valeur garantie pour MAX

— β : meilleure valeur garantie pour MIN

Condition d'élagage.

— Si un nœud MAX trouve $val \geq \beta$, MIN ne l'acceptera jamais $\Rightarrow$ coupure.

— Si un nœud MIN trouve $val \leq \alpha$, MAX ne l'acceptera jamais $\Rightarrow$ coupure.

Algorithm 2 Algorithme Alpha-Bêta

Require: Arbre a, profondeur d, bornes α, β, booléen $estMax$
Ensure: Valeur d'évaluation optimale

if $a.nbEnfants = 0$ **or** $d = 0$ **then**
    return $evaluationEtape5(a.plateau)$
end if
if $estMax$ **then**
    $best \leftarrow -10^6$
    for $i = 0$ à $a.nbEnfants - 1$ **do**
        $val \leftarrow alphaBeta(a.enfants[i], d-1, \alpha, \beta, false)$
        $best \leftarrow \max(best, val)$
        $\alpha \leftarrow \max(\alpha, best)$
        if $\alpha \geq \beta$ **then**
            return $best$ ▷ Élagage
        end if
    end for
else
    $best \leftarrow 10^6$
    for $i = 0$ à $a.nbEnfants - 1$ **do**
        $val \leftarrow alphaBeta(a.enfants[i], d-1, \alpha, \beta, true)$
        $best \leftarrow \min(best, val)$
        $\beta \leftarrow \min(\beta, best)$
        if $\alpha \geq \beta$ **then**
            return $best$ ▷ Élagage
        end if
    end for
end if
return $best$

3.6 Minimax vs Alpha-Bêta : protocole et résultats

Objectif. Comparer Minimax et Alpha-Bêta sur :

— nombre de nœuds explorés

— temps d'exécution

avec une profondeur variant de 1 à 7.

Conditions.

— 3 plateaux types : début, milieu, fin de partie

— même fonction d'évaluation

— exécution automatisée en C

— export en `.csv` puis analyse Python

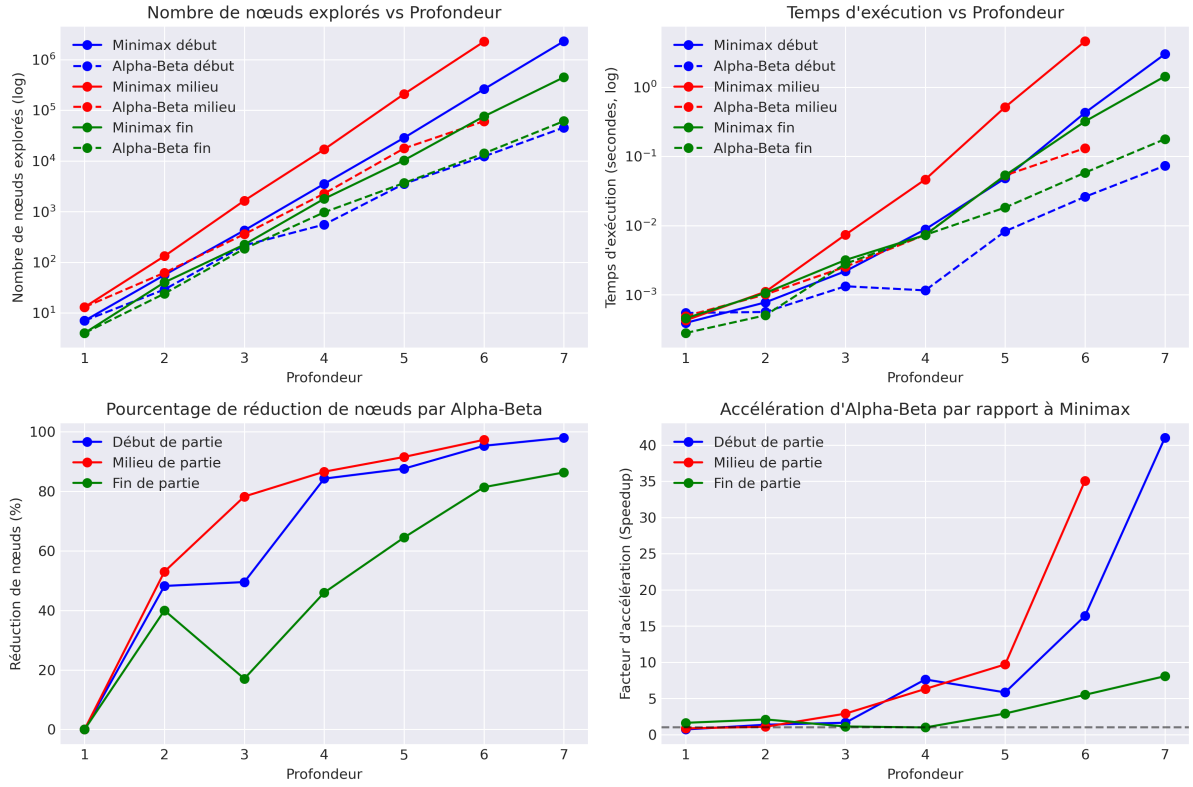

FIGURE 2 – Comparaisons Minimax vs Alpha-Bêta sur trois phases de jeu.

4 Optimisation mémoire

4.1 Motivation

L'exploration d'un arbre complet consomme énormément de mémoire. Il est donc nécessaire d'optimiser la représentation des états pour permettre une recherche plus profonde.

4.2 Approche retenue

— Suppression de l'arbre complet : un seul plateau est utilisé.

— Pour chaque coup simulé : sauvegarde uniquement des modifications via `HistoriqueCoup`.

— On applique un coup puis on l'annule grâce à l'historique.

4.3 Gestion de la mémoire dynamique

Nous mettons à jour manuellement trois compteurs :

— mémoire actuelle utilisée

— mémoire maximale pendant un coup

— mémoire totale cumulée sur une partie

4.4 Difficultés rencontrées et solutions

Difficultés.

— Mesure complexe de la mémoire sans arbre global

— Gestion fine des allocations (coups possibles, historique)

— Risque d'erreurs si l'historique n'est pas remis à zéro correctement

Solutions.

— Suivi manuel précis de chaque `malloc/free`

— Réinitialisation des compteurs à chaque appel IA

— Libération systématique après exploration

4.5 Approche classique vs optimisée : protocole

— Comparaison sur différentes profondeurs

— Mémoire moyenne par coup

— Mémoire totale sur une partie complète (30 plateaux)

Conclusion expérimentale. Dès la profondeur 3, la version optimisée réduit la mémoire utilisée de plus de 98%.

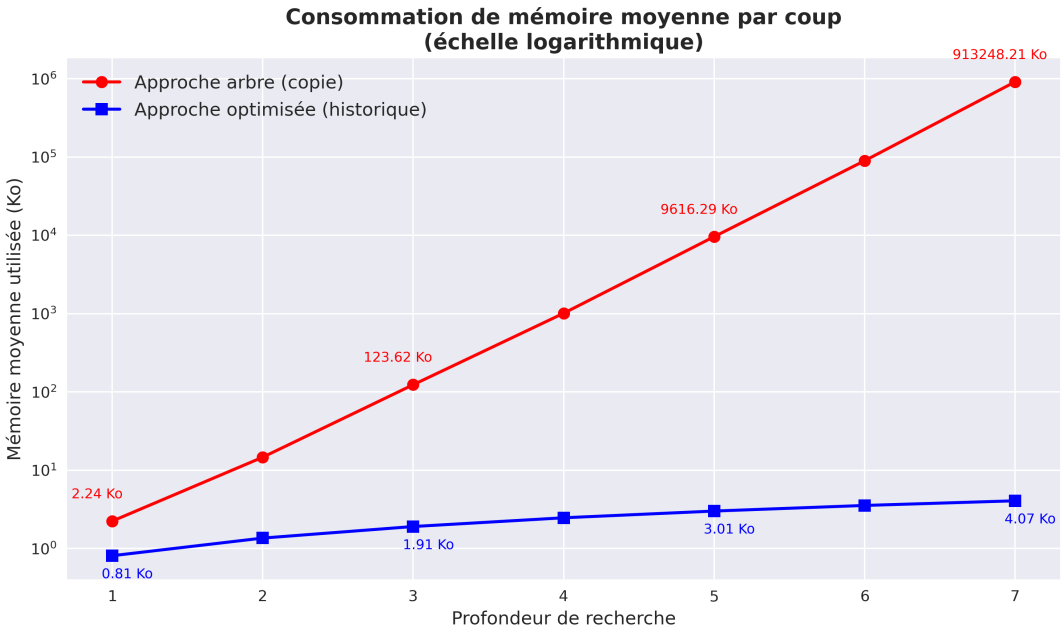

FIGURE 3 – Mémoire moyenne par coup : approche classique vs optimisée.

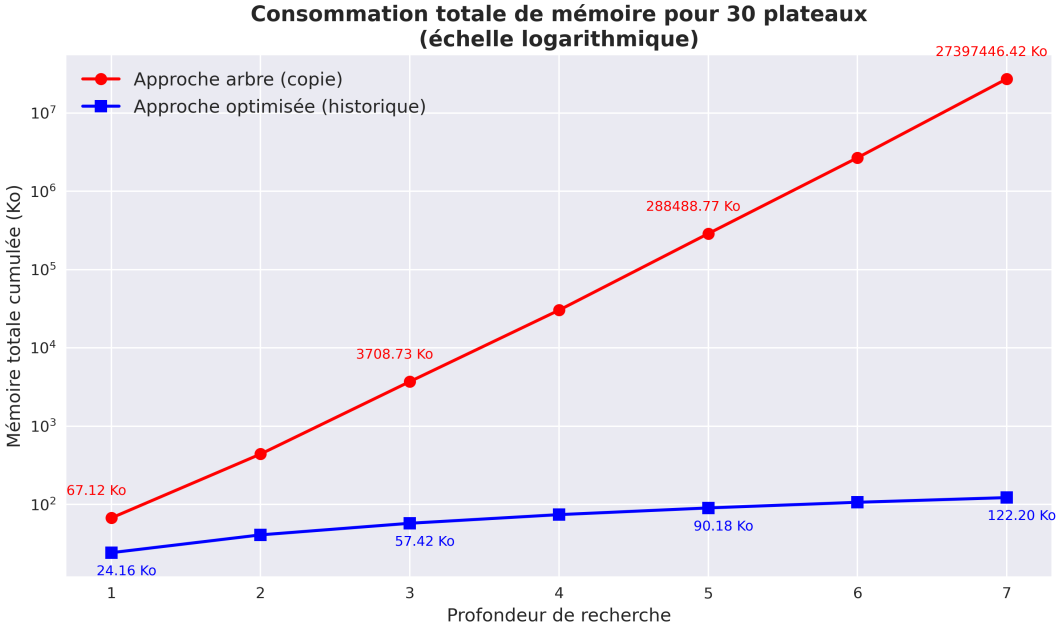

FIGURE 4 – Mémoire totale sur une partie complète.

5 Améliorations et initiatives

5.1 Améliorations de la fonction d'évaluation

Nouveaux critères ajoutés :

5.1.1 Mobilité

— Mobilité immédiate : nombre de coups légaux disponibles.
— Frontière : pions adjacents à des cases vides (vulnérables).
— Mobilité potentielle : capacité à restreindre les coups adverses.

5.1.2 Stabilité

— Stabilité absolue : pions impossibles à retourner (coins).
— Semi-stabilité : pions difficiles à retourner (bords/entourés).
— Propagation de stabilité : marquage récursif des pions stables.

5.1.3 Parité

— Cases vides paires : avantage au joueur ayant plus de pions.
— Cases vides impaires : avantage au joueur ayant moins de pions.

5.1.4 Disk-Square Tables

— Valeur spécifique par case selon la phase de jeu.
— Coins/bords stables fortement valorisés.
— Adjacent aux coins pénalisé si non sécurisé.

5.1.5 Pondération dynamique

— Début : priorité mobilité ($\times 5$) et positions stratégiques ($\times 2$).
— Milieu : équilibre mobilité ($\times 4$), stabilité ($\times 3$), ensembles ($\times 3$).
— Fin : priorité stabilité ($\times 6$) et parité ($\times 3$).
— Endgame : détection spécialisée des fins de partie.

5.2 Algorithme NegaScout

Principe. NegaScout est une amélioration d'Alpha-Bêta basée sur l'idée que le premier coup testé est souvent le meilleur.

Fenêtres nulles.

— Premier enfant exploré avec fenêtre complète $[\alpha, \beta]$.
— Suivants explorés avec fenêtre nulle $[\alpha, \alpha + 1]$.
— Si dépassement, relance en fenêtre complète.

Intérêt. Réduction du nombre de nœuds, surtout si les coups sont bien triés.

Difficultés et solution. La version NegaMax s'est révélée délicate à adapter correctement. Nous avons adopté une version Min/Max plus lisible et stable.

Algorithm 3 Algorithme NegaScout (convention Min/Max)

Require: Arbre a, profondeur d, bornes α, β, booléen $estMax$
Ensure: Valeur optimale

if $a.nbEnfants = 0$ **or** $d = 0$ **then**
  return $evaluationPonderee7(a.plateau)$
end if
if $estMax$ **then**
  $best \leftarrow -10^6$
  for $i = 0$ à $a.nbEnfants - 1$ **do**
    if $i = 0$ **then**
      $val \leftarrow negaScout(a.enfants[i], d-1, \alpha, \beta, false)$
    else
      $val \leftarrow negaScout(a.enfants[i], d-1, \alpha, \alpha+1, false)$
      if $val > \alpha$ **and** $val < \beta$ **then**
        $val \leftarrow negaScout(a.enfants[i], d-1, val, \beta, false)$
      end if
    end if
    $best \leftarrow \max(best, val)$
    $\alpha \leftarrow \max(\alpha, best)$
    if $\alpha \geq \beta$ **then**
      return $best$
    end if
  end for
else
  $best \leftarrow 10^6$
  for $i = 0$ à $a.nbEnfants - 1$ **do**
    if $i = 0$ **then**
      $val \leftarrow negaScout(a.enfants[i], d-1, \alpha, \beta, true)$
    else
      $val \leftarrow negaScout(a.enfants[i], d-1, \beta-1, \beta, true)$
      if $val > \alpha$ **and** $val < \beta$ **then**
        $val \leftarrow negaScout(a.enfants[i], d-1, \alpha, val, true)$
      end if
    end if
    $best \leftarrow \min(best, val)$
    $\beta \leftarrow \min(\beta, best)$
    if $\alpha \geq \beta$ **then**
      return $best$
    end if
  end for
end if
return $best$

5.3 Tri des coups : tests

Contexte. Après l'optimisation mémoire de NegaScout, nous avons étudié l'importance du *move ordering*. La fonction d'évaluation complète étant coûteuse, un tri dynamique est trop lent.

Variantes comparées.

— NegaScout sans tri.

— Tri statique (évaluation simple, sans jouer le coup).

— Tri dynamique (évaluation complète + annulation).

Protocole. Simulation de 30 parties ; pour chaque profondeur :

— moyenne du nombre de nœuds explorés

— moyenne du temps de réflexion

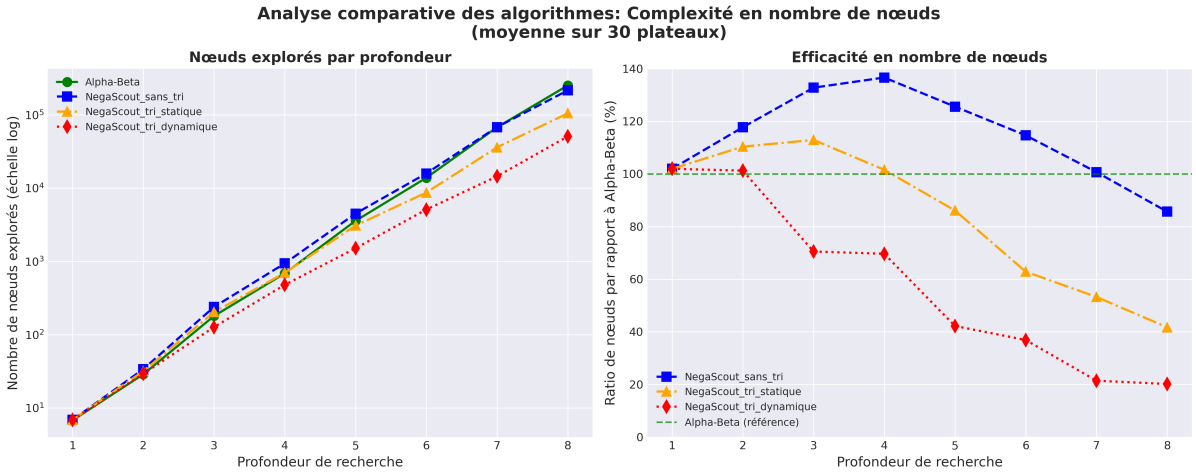

FIGURE 5 – Résultats : nombre de nœuds explorés selon la variante de tri.

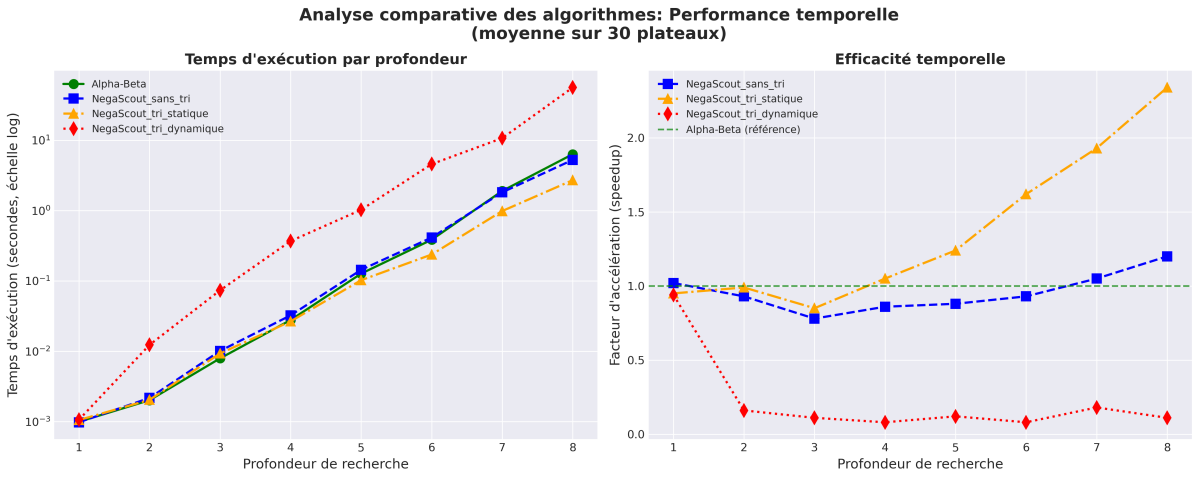

FIGURE 6 – Résultats : temps de réflexion selon la variante de tri.

Conclusion. Le tri statique réduit efficacement les nœuds explorés, tandis que le tri dynamique, bien qu'un peu meilleur en élagage, est trop coûteux. Nous adoptons donc le tri statique comme meilleur compromis.

5.4 Gestion du Endgame

5.4.1 Détection automatique

La phase finale est détectée par :

— faible nombre de cases vides (≤ 14) ou très faible mobilité (≤ 8 coups)

— domination des coins (≥ 3 coins occupés) ou ≥ 50 pions joués

en s'appuyant sur les seuils d'exhaustivité de Michael Buro.

5.4.2 Objectif en Endgame

Contrairement au midgame :

— l'IA explore exhaustivement toutes les suites possibles

— calcule exactement le vainqueur

— optimise le score final

5.4.3 Recherche exhaustive et heuristique

— NegaScout adapté à l'Endgame avec profondeur dynamique.

— Évaluation exacte dès que la partie est terminée.

— Heuristique *Fastest-First* : priorité aux coups minimisant la mobilité adverse et sécurisant les coins.

5.5 Pistes futures

— **Parallélisation** par threads pour explorer les branches en parallèle

— **Évaluation par apprentissage** : remplacer les règles manuelles par un modèle entraîné sur des parties passées.

6 Conclusion

Nous avons construit une IA Reversi robuste par développement incrémental. Les algorithmes de recherche ont été progressivement optimisés (Minimax → Alpha-Bêta → NegaScout), et l'optimisation mémoire a permis d'augmenter significativement la profondeur de recherche. L'évaluation heuristique enrichie (mobilité, stabilité, parité, tables) améliore la qualité de jeu, tandis que le tri statique offre un compromis efficace entre nœuds explorés et temps de calcul. Des perspectives prometteuses restent ouvertes, notamment la parallélisation et l'apprentissage automatique.

7 Ressources

- Documentation WZebra : http://www.radagast.se/othello/
- Fédération Française d'Othello : https://www.ffothello.org/informatique/algorithmes/
- Matplotlib : https://matplotlib.org/
- Publications de Michael Buro : https://skatgame.net/mburo/publications.html