Projet d'Intelligence Artificielle

Implémentation et étude de stratégies pour le jeu *Dots & Boxes*

Alexis VÕ — Thanh Phát [redacted] — Tianqi [redacted] — Rakul [redacted]
LDD3 IM, Université Paris-Saclay

20 juillet 2026

Table des matières

1 Introduction **2**

2 Modélisation du jeu **2**
2.1 Structure du plateau . . . 2
2.2 Architecture logicielle . . . 2
2.3 Schéma de représentation du plateau . . . 2
2.4 Représentation des actions . . . 3
2.5 Formalisation mathématique . . . 3

3 Règles du jeu **4**

4 Analyse de l'arbre de jeu **4**
4.1 Facteur de branchement . . . 4
4.2 Profondeur maximale . . . 4
4.3 Conséquence algorithmique . . . 5
4.4 Illustration simplifiée d'un arbre de décision . . . 5
4.5 Observations stratégiques préliminaires . . . 5

5 Stratégies de jeu **6**
5.1 Stratégies simples . . . 6
5.1.1 Comparaison expérimentale Random / Glouton . . . 6
5.2 Stratégies IA . . . 6
5.2.1 Minimax . . . 6
5.2.2 Alpha-Beta . . . 8
5.2.3 MCTS . . . 9
5.3 Stratégie Expert . . . 10
5.3.1 Motivation . . . 10
5.3.2 Bitboard . . . 10
5.3.3 Hachage de Zobrist . . . 11
5.3.4 Tables de transposition . . . 11
5.3.5 Fonction d'évaluation et théorie de la parité . . . 12
5.3.6 Point d'entrée de la décision : `selectAction` . . . 13
5.3.7 Bilan . . . 13

6 Tests et validation **14**

7 Conclusion **15**

1 Introduction

Le projet présenté dans ce rapport consiste à implémenter le jeu *Dots and Boxes*, un jeu combinatoire à information parfaite opposant deux joueurs. Dans ce jeu, les joueurs tracent alternativement des segments sur une grille, dans le but de compléter des cases. Lorsqu'un joueur parvient à fermer une case, celle-ci lui est attribuée et il bénéficie d'un tour supplémentaire. La partie se termine lorsque toutes les cases ont été capturées, et le vainqueur est alors le joueur possédant le plus grand nombre de cases.

Ce jeu nous a permis de mettre en application les algorithmes de prise de décision vus en cours. En effet, il s'agit d'un problème typique de théorie des jeux, dans lequel chaque action influence directement les possibilités futures.

L'objectif de ce projet est triple. Il s'agit tout d'abord de proposer une modélisation rigoureuse du jeu, en définissant précisément les structures de données nécessaires à sa représentation. Ensuite, plusieurs stratégies de jeu sont implémentées, allant de méthodes simples à des approches plus avancées issues de l'intelligence artificielle. Enfin, ces différentes stratégies sont comparées afin d'analyser leurs performances respectives.

Ce travail mobilise plusieurs notions fondamentales du cours, notamment l'exploration d'arbres de jeu, les algorithmes Minimax et Alpha-Beta, les méthodes probabilistes telles que le Monte Carlo Tree Search (MCTS), ainsi que l'utilisation d'heuristiques pour guider la prise de décision.

2 Modélisation du jeu

2.1 Structure du plateau

Le plateau de jeu est représenté par une classe `Board`, qui constitue le cœur de la modélisation. Celui-ci est défini par un nombre de lignes n et un nombre de colonnes m, correspondant au nombre de points de la grille. Le nombre de cases du jeu est donc $(n-1)(m-1)$.

Afin de représenter les coups possibles, nous avons utilisé deux structures distinctes : l'une pour les arêtes horizontales et l'autre pour les arêtes verticales. Ce choix permet de simplifier la gestion des actions et de distinguer clairement les différents types de segments.

Par ailleurs, chaque case du plateau est associée à un propriétaire, stocké dans une structure dédiée. Une valeur de -1 indique qu'une case n'a pas encore été capturée, tandis que les valeurs 0 et 1 correspondent respectivement aux deux joueurs.

Cette modélisation permet ainsi de représenter de manière fidèle l'état du jeu à tout instant. Les opérations de mise à jour sont également facilitées.

2.2 Architecture logicielle

L'architecture du projet sépare clairement l'état du jeu, les actions et la logique de décision. La classe `Board` stocke l'état courant du plateau : segments horizontaux, segments verticaux et propriétaires des cases. La classe `Action` représente un coup élémentaire, défini par une orientation et des coordonnées.

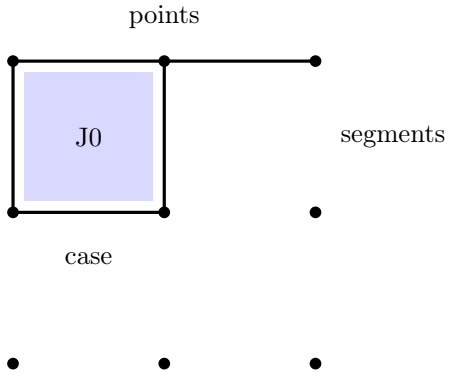
Les joueurs sont modélisés par l'interface `Player`, dont le rôle est de retourner une action à partir du plateau courant. Les stratégies automatiques implémentent l'interface `ActionStrategy`, ce qui permet de changer l'algorithme de décision sans modifier le moteur du jeu.

La transition principale est réalisée par `Board.apply(Action, int)`, qui trace le segment, détecte les cases fermées et les attribue au joueur courant. La classe `Referee` maintient les scores et applique les pénalités, tandis que `DotsBoxesGame` gère la boucle de jeu, le changement de joueur et la règle du tour supplémentaire lorsqu'un joueur ferme une case.

2.3 Schéma de représentation du plateau

La figure suivante illustre le principe général du jeu sur une petite grille. Les points représentent les sommets, les segments horizontaux et verticaux représentent les coups possibles, et les cases

correspondent aux zones susceptibles d'être capturées par les joueurs.



Dans notre implémentation, le plateau n'est pas stocké sous forme graphique, mais à l'aide de structures booléennes distinguant les arêtes horizontales et verticales. Cette séparation présente un intérêt pratique : elle simplifie la vérification des coups valides ainsi que la détection des cases fermées après chaque action.

2.4 Représentation des actions

Une action est modélisée par la classe `Action`, qui encapsule les informations nécessaires à la description d'un coup. Chaque action est caractérisée par un type, qui peut être `HORIZONTAL` ou `VERTICAL`, ainsi que par une position (i, j) indiquant l'emplacement de l'arête sur le plateau.

Cette abstraction permet de manipuler les coups de manière uniforme indépendamment de leur orientation. Elle simplifie l'implémentation des différentes stratégies de jeu qui peuvent ainsi raisonner sur un ensemble homogène d'actions.

2.5 Formalisation mathématique

Le jeu peut être formalisé comme un jeu déterministe à deux joueurs, à information parfaite. Les joueurs sont :

$$P = \{J_0, J_1\}.$$

Un état du jeu est défini par :

$$s = (H, V, B, p),$$

où H est l'ensemble des segments horizontaux déjà tracés, V l'ensemble des segments verticaux déjà tracés, B l'attribution des cases aux joueurs ou à la valeur -1 lorsqu'elles sont encore libres, et $p \in P$ le joueur actif.

L'ensemble des actions légales depuis un état s, noté $A(s)$, correspond aux segments horizontaux ou verticaux encore non tracés et dont les coordonnées respectent les bornes du plateau. Dans le code, cet ensemble est construit par `Board.getAvailableActions()`.

La fonction de transition est :

$$T : S \times A(s) \to S.$$

Elle correspond à l'application d'un coup par `Board.apply(Action, int)`. Si le coup ferme une ou plusieurs cases, celles-ci sont attribuées au joueur courant et le joueur actif reste le même. Sinon, le joueur actif devient l'adversaire. Cette règle est essentielle car le jeu n'est pas strictement alterné.

Une fonction d'utilité terminale naturelle est la différence de score :

$$U(s) = \text{score}(J_1) - \text{score}(J_0).$$

Cette formalisation justifie l'utilisation d'algorithmes classiques d'intelligence artificielle pour les jeux, comme Minimax et Alpha-Beta.

3 Règles du jeu

L'application d'un coup est réalisée à l'aide de la méthode suivante :

```
int apply(Action action, int playerId)
```

Cette méthode joue un rôle central dans la logique du jeu. Elle permet d'appliquer une action donnée, de vérifier sa validité, et de déterminer le nombre de cases éventuellement fermées par ce coup. En cas de fermeture d'une ou plusieurs cases, celles-ci sont attribuées au joueur courant, et son score est mis à jour en conséquence.

Plusieurs règles essentielles doivent être respectées pour garantir la cohérence du jeu :

Règles importantes

- Dans `Board.apply`, une action invalide entraîne une exception, ce qui permet de détecter immédiatement les erreurs au niveau du plateau.
- Dans la boucle de jeu, `DotsBoxesGame` vérifie les actions invalides ou nulles avant l'application du coup. Dans notre implémentation, un coup invalide entraîne une pénalité de -1 point et la perte du tour via `Referee.applyInvalidMovePenalty`.
- Lorsqu'un joueur ferme une case, il bénéficie d'un tour supplémentaire.
- Une case ne peut être capturée qu'une seule fois, ce qui garantit l'intégrité des scores.

Le sujet indique une pénalité égale au nombre de lignes du plateau. Notre implémentation applique une pénalité fixe de -1, ce qui est cohérent avec les tests unitaires actuels mais constitue une différence par rapport à l'énoncé.

4 Analyse de l'arbre de jeu

L'un des points centraux du projet est l'étude de l'arbre de jeu associé à *Dots and Boxes*. Chaque sommet de cet arbre correspond à un état possible du plateau, et chaque arête correspond à une action légale jouée par l'un des deux joueurs.

4.1 Facteur de branchement

Si le plateau contient n lignes de points et m colonnes de points, alors le nombre total de segments jouables est donné par :

$$n(m-1) + (n-1)m = 2nm - n - m.$$

En effet, il y a $n(m-1)$ segments horizontaux et $(n-1)m$ segments verticaux.

Par exemple, pour une grille de 4×4 points, il y a $4(4-1) + (4-1)4 = 24$ segments jouables au début de la partie. Cela correspond au facteur de branchement maximal initial.

Au début de la partie, ce nombre correspond au facteur de branchement maximal. Ensuite, ce facteur décroît progressivement au fur et à mesure que des segments sont tracés.

4.2 Profondeur maximale

Chaque coup ajoute exactement un segment au plateau. Par conséquent, la profondeur maximale théorique d'une partie est égale au nombre total de segments :

$$D_{\max} = 2nm - n - m.$$

En pratique, cette profondeur peut être légèrement plus délicate à exploiter dans l'analyse, car un joueur peut rejouer après avoir fermé une case. Cela signifie que l'alternance des joueurs n'est pas strictement régulière, ce qui rend l'arbre de jeu moins simple qu'un arbre minimax standard.

Dans l'implémentation de Minimax, cette propriété impose de ne pas changer systématiquement de joueur à chaque niveau de l'arbre. Si un coup ferme une case, le nœud fils correspond encore au même joueur actif ; sinon, le tour passe à l'adversaire. Cette distinction est nécessaire pour évaluer correctement les séquences de captures.

4.3 Conséquence algorithmique

Cette croissance rapide du nombre d'états rend impossible une exploration exhaustive dès que la taille du plateau augmente. C'est précisément ce qui motive l'utilisation :

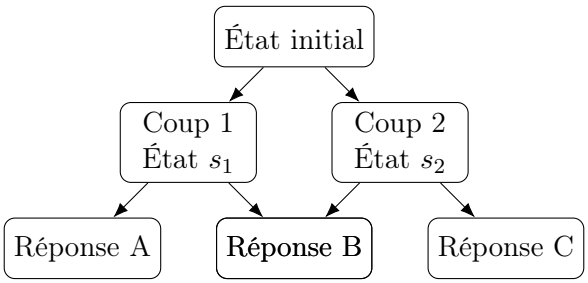
— d'une profondeur limitée pour Minimax,

— d'un élagage avec Alpha-Beta,

— ou encore d'approches probabilistes comme MCTS.

4.4 Illustration simplifiée d'un arbre de décision

La figure suivante donne une représentation très simplifiée d'un arbre de jeu. Chaque nœud correspond à un état du plateau, et chaque branche à un coup possible.



Dans la réalité, un tel arbre devient très rapidement volumineux, car le nombre d'actions disponibles reste important pendant une grande partie de la partie. C'est pourquoi la qualité des heuristiques et les optimisations de recherche jouent un rôle essentiel.

4.5 Observations stratégiques préliminaires

Des parties jouées en mode humain contre humain permettent de mettre en évidence plusieurs situations importantes. La fermeture d'une case donne immédiatement un avantage local, car le joueur gagne un point et conserve la main. Cependant, cet avantage peut devenir trompeur lorsque le coup joué ouvre une chaîne de cases à l'adversaire.

Une position particulièrement dangereuse est la case possédant déjà trois côtés tracés. Le joueur qui laisse une telle case offre généralement un point immédiat à l'adversaire, ainsi qu'un tour supplémentaire. Lorsque plusieurs cases de ce type sont reliées, elles forment des chaînes : le joueur qui déclenche la chaîne peut parfois être obligé de concéder plusieurs cases successives.

Cette observation explique pourquoi une stratégie purement gloutonne, qui ferme toujours une case dès que possible, n'est pas nécessairement optimale. Une stratégie plus forte doit anticiper les réponses adverses, éviter de créer trop tôt des cases à trois côtés et raisonner sur les chaînes de fin de partie. Ces constats préparent l'introduction de Minimax, d'Alpha-Beta et des stratégies expertes.

Nous avons également comparé ces observations avec le jeu en ligne proposé dans l'énoncé. Le comportement observé est similaire : les coups apparemment avantageux localement peuvent devenir défavorables lorsqu'ils ouvrent une chaîne, tandis que les positions avec trois côtés tracés sont exploitées immédiatement par l'adversaire. Cette comparaison confirme que les phénomènes de chaînes et de contrôle de l'initiative doivent être pris en compte par nos stratégies automatiques.

5 Stratégies de jeu

5.1 Stratégies simples

Random La stratégie `RandomActionStrategy` choisit aléatoirement une action parmi les coups valides retournés par `Board.getAvailableActions()`. Elle sert de stratégie de référence minimale, car elle ne tient compte ni du score courant, ni des cases presque fermées.

FirstValid La stratégie `FirstValidActionStrategy` retourne simplement la première action disponible. Elle est surtout utile comme stratégie déterministe de base pour les tests, car son comportement est prévisible.

Glouton La stratégie `GloutonActionStrategy` applique une heuristique locale plus forte :

- elle privilégie les coups qui ferment immédiatement une ou plusieurs cases ;
- à défaut, elle cherche un coup qui n'offre pas d'opportunité immédiate à l'adversaire, c'est-à-dire un coup qui évite de créer une case avec trois côtés ;
- si aucun coup sûr n'est disponible, elle choisit aléatoirement parmi les coups restants.

Cette stratégie reste simple car elle ne construit pas d'arbre de recherche. Elle permet toutefois de montrer qu'une décision locale informée est déjà nettement plus efficace qu'un choix uniformément aléatoire, tout en conservant certaines limites dans les configurations de chaînes.

5.1.1 Comparaison expérimentale Random / Glouton

Nous avons comparé les stratégies `RandomActionStrategy` et `GloutonActionStrategy` sur 200 parties avec une grille de 3×3 points, en utilisant la même configuration expérimentale pour les deux joueurs.

Stratégie	Victoires	Nuls	Défaites	Score total	Score moyen
Random	22	62	116	248	1.240
Glouton	116	62	22	552	2.760

L'écart moyen est de 1.520 case par partie en faveur de la stratégie gloutonne. Ce résultat confirme qu'une stratégie capable de fermer immédiatement les cases disponibles est beaucoup plus performante qu'un choix aléatoire.

Cependant, cette stratégie reste locale. Nous avons identifié une position de milieu de partie sur une grille de 4×4 points où `GloutonActionStrategy` choisit `HORIZONTAL(1,0)`, tandis qu'une recherche Alpha-Beta plus profonde choisit `VERTICAL(2,2)`. Cette divergence montre que la stratégie gloutonne peut manquer des conséquences futures liées aux chaînes et aux cases à trois côtés. Elle motive donc l'introduction de stratégies adversariales comme Minimax.

5.2 Stratégies IA

5.2.1 Minimax

L'algorithme Minimax constitue une approche classique pour les jeux à deux joueurs à somme nulle. Il repose sur l'exploration récursive de l'arbre des coups possibles, en alternant entre deux types de nœuds : les nœuds maximisants, correspondant au joueur courant, et les nœuds minimisants, correspondant à l'adversaire.

Afin d'évaluer les positions intermédiaires, nous utilisons une heuristique simple plutôt qu'une différence de score brute. Cette fonction d'évaluation retourne une valeur normalisée entre -1 et 1, où -1 correspond à une victoire certaine du joueur 0 et 1 à une victoire certaine du joueur 1.

Lorsque la partie est terminée, la fonction retourne directement le résultat final. Sinon, elle combine :

— la différence de score entre les deux joueurs,

— ainsi qu'un terme heuristique prenant en compte les cases presque complètes (cases avec trois côtés), qui représentent des opportunités immédiates de capture.

Les cases à trois côtés sont légèrement pénalisées ou favorisées selon le joueur courant, afin de refléter leur impact stratégique. Le score final est ensuite borné dans l'intervalle $[-1, 1]$.

Cependant, Minimax présente une complexité exponentielle, ce qui limite son utilisation à des profondeurs de recherche relativement faibles.

Pseudo-code simplifié de Minimax Le principe de Minimax peut être résumé par le pseudo-code suivant :

```
fonction Minimax(etat, profondeur, joueur):
    si etat terminal ou profondeur = 0:
        retourner Evaluation(etat)

    si joueur = MAX:
        meilleur <- -infini
    sinon:
        meilleur <- +infini

    pour chaque action possible:
        successeur, casesFermees <- appliquer(etat, action, joueur)

        si casesFermees > 0:
            joueurSuivant <- joueur
            profondeurSuivante <- profondeur
        sinon:
            joueurSuivant <- adversaire(joueur)
            profondeurSuivante <- profondeur - 1

        valeur <- Minimax(successeur, profondeurSuivante, joueurSuivant)

        si joueur = MAX:
            meilleur <- max(meilleur, valeur)
        sinon:
            meilleur <- min(meilleur, valeur)

    retourner meilleur
```

Dans notre cas, une difficulté supplémentaire apparaît : si un joueur ferme une case, il rejoue immédiatement. Notre implémentation conserve alors le même joueur actif et ne diminue pas la profondeur de recherche pour ce tour supplémentaire ; dans le cas contraire, le tour passe à l'adversaire et la profondeur est décrémentée.

Analyse expérimentale de la profondeur Nous avons ensuite mesuré le temps moyen de décision de Minimax pour plusieurs profondeurs de recherche. Les résultats suivants ont été obtenus sur des positions générées automatiquement, en vérifiant à chaque fois que le coup retourné restait valide.

Plateau	Profondeur	Temps moyen (ms)	Coups valides
3×3 points	1	0.138	12/12
3×3 points	2	0.292	12/12
3×3 points	3	1.491	12/12
4×4 points	1	0.489	12/12
4×4 points	2	6.890	12/12
4×4 points	3	391.997	12/12

Ces mesures montrent que le temps de décision augmente rapidement avec la profondeur, surtout lorsque la taille du plateau augmente. Sur une grille de 3×3 points, la profondeur 3 reste très abordable. En revanche, sur une grille de 4×4 points, la profondeur 3 devient déjà beaucoup plus coûteuse. Cela illustre le compromis classique entre qualité de décision et temps de calcul.

5.2.2 Alpha-Beta

L'algorithme Alpha-Beta constitue une amélioration directe de Minimax. Il permet de réduire significativement le nombre de nœuds explorés en éliminant les branches qui ne peuvent pas influencer le résultat final.

Ce mécanisme d'élagage repose sur l'utilisation de deux bornes, notées α et β, qui permettent de détecter précocement les situations non pertinentes. Malgré cette optimisation, l'algorithme conserve les mêmes garanties que Minimax en termes de résultat.

Pseudo-code simplifié de Alpha-Beta L'algorithme Alpha-Beta reprend la logique de Minimax, tout en évitant l'exploration de certaines branches manifestement inutiles.

```
fonction AlphaBeta(etat, profondeur, alpha, beta, joueur):
    si etat terminal ou profondeur = 0:
        retourner Evaluation(etat)

    si joueur = MAX:
        valeur <- -infini
    sinon:
        valeur <- +infini

    pour chaque action possible:
        successeur, casesFermees <- appliquer(etat, action, joueur)

        si casesFermees > 0:
            joueurSuivant <- joueur
            profondeurSuivante <- profondeur
        sinon:
            joueurSuivant <- adversaire(joueur)
            profondeurSuivante <- profondeur - 1

        score <- AlphaBeta(successeur, profondeurSuivante,
                           alpha, beta, joueurSuivant)

        si joueur = MAX:
            valeur <- max(valeur, score)
            alpha <- max(alpha, valeur)
        sinon:
            valeur <- min(valeur, score)
            beta <- min(beta, valeur)

        si alpha >= beta:
            couper la branche

    retourner valeur
```

Cette amélioration ne modifie pas le résultat théorique obtenu, mais réduit en pratique le nombre de nœuds visités, parfois de manière très significative.

Comparaison expérimentale avec Minimax Nous avons comparé Minimax et Alpha-Beta sur des positions générées automatiquement, avec les mêmes profondeurs de recherche. Les mesures portent sur le temps moyen de décision, le taux de coups identiques et le nombre moyen de nœuds visités par Alpha-Beta.

Plateau	Prof.	Minimax (ms)	Alpha-Beta (ms)	Même coup	Nœuds AB
3×3	1	0.101	0.311	0.67	10.3
3×3	2	0.372	0.368	0.87	50.1
3×3	3	1.005	0.303	0.47	143.3
4×4	1	0.466	0.602	0.33	32.5
4×4	2	5.996	0.438	0.53	211.9
4×4	3	381.616	1.156	0.53	1144.2

Ces résultats montrent que l'élagage Alpha-Beta devient particulièrement efficace lorsque la profondeur augmente. Sur une grille de 4×4 points à profondeur 3, le temps moyen passe de 381.616 ms pour Minimax à 1.156 ms pour Alpha-Beta. Le taux de coups identiques n'est pas toujours égal à 1, ce qui s'explique par les choix de départage entre actions de valeur proche ou égale, mais les tests unitaires vérifient la cohérence des deux approches sur des positions représentatives.

5.2.3 MCTS

Le Monte Carlo Tree Search (MCTS) adopte une approche différente, basée sur des simulations aléatoires. À partir d'un état donné, un grand nombre de parties sont simulées aléatoirement, et la qualité d'un coup est estimée à partir de la fréquence de victoire observée.

Cette méthode est particulièrement adaptée aux espaces de recherche très vastes, dans lesquels les approches déterministes deviennent trop coûteuses.

Principe général Le MCTS repose classiquement sur quatre phases :

1. **Sélection** : on descend dans l'arbre selon un critère d'exploration/exploitation ;
2. **Expansion** : on ajoute un nouveau nœud correspondant à un coup encore inexploré ;
3. **Simulation** : on simule aléatoirement une partie jusqu'à un état terminal ;
4. **Rétropropagation** : on met à jour les statistiques des nœuds visités.

Politique de simulation (rollout) Dans notre implémentation, les simulations ne sont pas entièrement aléatoires. Nous utilisons une politique de rollout guidée, inspirée d'une stratégie gloutonne simple, afin d'améliorer la qualité des estimations.

À chaque étape de la simulation :

— si un coup permet de fermer immédiatement une ou plusieurs cases, il est sélectionné en priorité

— sinon, on privilégie les coups dits *sûrs*, c'est-à-dire ceux qui n'offrent pas à l'adversaire la possibilité de fermer une case au tour suivant (en évitant notamment de créer des cases à trois côtés)

— si aucun coup sûr n'est disponible, un coup est choisi aléatoirement parmi les coups restants

Cette politique permet de conserver une certaine part d'exploration tout en intégrant des heuristiques simples propres au jeu. En particulier, éviter de donner des opportunités immédiates à l'adversaire améliore significativement la qualité des simulations par rapport à des rollouts purement aléatoires.

Limite des simulations MCTS Dans notre implémentation, le nombre d'itérations du MCTS est paramétrable dans `MctsActionStrategy`. Dans la configuration utilisée depuis le moteur de jeu, il est fixé à $I = 1208$ itérations. Or, étant donné un facteur de branchement b et une profondeur maximale $D_{\max} = 2nm - n - m$, la taille de l'arbre de jeu est de l'ordre de $O(b^{D_{\max}})$. Ainsi, on a typiquement :

$$I \ll b^{D_{\max}}.$$

Par conséquent, seule une fraction très limitée de l'arbre est explorée, ce qui peut conduire à manquer certaines séquences tactiques précises. Cette limite motive l'utilisation d'heuristiques de rollout et, dans la stratégie experte, une combinaison avec Alpha-Beta.

5.3 Stratégie Expert

5.3.1 Motivation

L'objectif est de concevoir une IA experte en combinant les avantages du MCTS et de la recherche Alpha-Bêta. Toutefois, plusieurs limitations apparaissent dans nos implémentations actuelles.

Tout d'abord, malgré de l'élagage Alpha-Bêta, la profondeur atteignable reste limitée (typiquement 5 à 6 niveaux) sur des plateaux de taille modérée. Or, dans les phases de fin de partie, certaines séquences forcées peuvent être significativement plus longues. En particulier, des motifs caractéristiques du jeu, tels que le *double-cross*, nécessitent une anticipation profonde que cette limite empêche de capturer de manière fiable.

Ensuite, le coût computationnel du MCTS est élevé. Chaque itération de simulation nécessite la création d'une copie complète (*deep-copy*) du plateau afin de garantir l'indépendance des trajectoires explorées. Cette duplication répétée entraîne une consommation mémoire importante et sollicite fortement le *Garbage Collector (GC)* de Java, ce qui dégrade les performances globales.

Enfin, la fonction d'évaluation utilisée reste relativement simple. Bien qu'elle permette d'estimer grossièrement une position, elle manque de précision dans les situations intermédiaires complexes. En conséquence, les méthodes déterministes comme Alpha-Bêta ne disposent pas d'une heuristique suffisamment informative à profondeur limitée, et le MCTS ne peut pas pleinement exploiter des estimations fiables en dehors des états terminaux.

Ces limitations nous ont conduits à adopter une approche plus radicale, fondée sur une représentation et une manipulation alternatives du jeu, s'écartant des abstractions fournies par `Board.java` et `Action.java`, tout en les conservant comme points d'entrée et de sortie de la stratégie.

5.3.2 Bitboard

Inspirée des moteurs d'échecs modernes, nous avons introduit une représentation alternative du plateau : le *Bitboard*. L'idée centrale est d'encoder l'état de chaque segment (joué ou non) sous forme d'un seul bit, puis de regrouper ces bits dans des entiers de 64 bits (`long` en Java).

Considérons à titre d'exemple les segments verticaux d'un plateau 4×4 :

$$\begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix}$$

En lisant ces bits de gauche à droite et de haut en bas, on obtient la séquence `1010 0001 0101 1011`, stockée dans un unique entier `long`. L'état complet du plateau, comprenant les segments horizontaux, les segments verticaux et les cases capturées par chaque joueur, est ainsi encodé dans dix entiers `long` au total :

```
final long[] hBits = new long[3]; // segments horizontaux (3 x 64 >= 132)
final long[] vBits = new long[3]; // segments verticaux
final long[] box0  = new long[2]; // cases capturées par le joueur 0
final long[] box1  = new long[2]; // cases capturées par le joueur 1
```

Pour un plateau de taille maximale 12×12, cela représente 264 segments, soit au plus $3 \times 64 = 192$ bits par orientation. L'intégralité de l'état du jeu tient donc dans $6 \times 8 = 48$ octets, ce qui permet au processeur de maintenir le plateau en cache L1/L2 pendant toute la durée de la recherche.

Cette représentation apporte trois avantages concrets.

Opérations bit-à-bit Toutes les opérations : vérifier si un segment a été joué, compter les segments d'une case, ou détecter une capture, etc. se réduisent à des opérations `AND`, `OR` et `XOR`, qui sont très rapides et optimisées pour le processeur.

Appliquer et annuler un coup en $O(1)$ Plutôt que de copier l'intégralité du plateau à chaque nœud de l'arbre de recherche, nous maintenons une pile d'annulation (*undo stack*). Appliquer un coup consiste à poser un bit et à empiler les informations nécessaires à son annulation (segment joué, cases éventuellement capturées, joueur). Annuler ce coup consiste à effacer ce bit et à dépiler tout en $O(1)$ sans aucune allocation mémoire. Une seule instance de plateau est ainsi maintenue en mémoire pendant toute la recherche, éliminant les copies profondes et la pression sur le *GC*.

Localité de cache Les tableaux `hBits`, `vBits`, `box0` et `box1` (ainsi que tous les autres tableaux auxiliaires utilisés pour les calculs) sont alloués une seule fois et réutilisés jusqu'à la fin de la partie. Leur contiguïté en mémoire favorise la localité spatiale de cache et éviter l'allocation constante de mémoire pendant la phase critique de la recherche.

Ces propriétés constituent le socle sur lequel reposent les deux mécanismes suivants.

5.3.3 Hachage de Zobrist

Pour éviter de réévaluer des positions déjà rencontrées, il est nécessaire d'identifier chaque état du jeu par une clé compacte. Nous utilisons le *hachage de Zobrist*, une technique classique des moteurs d'échecs, particulièrement bien adaptée à la représentation par bitboard.

Au démarrage, on génère aléatoirement un entier 64 bits indépendant pour chaque segment du plateau (horizontal ou vertical). La clé de hachage H est initialisée à zéro, puis mise à jour à chaque coup par une opération XOR :

$$H \leftarrow H \oplus r_s$$

où r_s est le nombre aléatoire associé au segment s que l'on vient de jouer. Annuler ce coup revient à appliquer à nouveau le même XOR, ce qui est une propriété fondamentale de cette opération. La clé est ainsi maintenue à jour en $O(1)$ lors de chaque `apply()` et `undo()`, sans recalcul complet.

Une subtilité importante concerne le tour de jeu : deux positions identiques mais appartenant à des joueurs différents sont stratégiquement distinctes. Pour les différencier, on XOR la clé courante avec une constante `PLAYER_ZOBRIST` lorsque c'est au joueur 1 de jouer :

```
long hash = fb.zobrist ^ (playerId == 1 ? FastBoard.PLAYER_ZOBRIST : 0L);
```

Ce schéma ne garantit pas l'absence de collision, mais le taux observé est négligeable en pratique.

5.3.4 Tables de transposition

La table de transposition est la structure de données qui exploite le hachage de Zobrist pour mémoriser les résultats de l'exploration. Nous maintenons deux tables distinctes, chacune indexée par la clé de Zobrist via un masque bit-à-bit (adressage modulo puissance de deux, sans division) :

— `ABTranspositionTable` : stocke la valeur minimax, la profondeur, le type de borne (exacte, inférieure, supérieure) et le meilleur coup. Lors d'une consultation, si la borne permet un élagage immédiat la valeur est retournée directement ; sinon elle resserre la fenêtre $[\alpha, \beta]$ courante. La politique de remplacement est *profondeur d'abord* : une entrée n'est écrasée que par un enregistrement de profondeur supérieure ou égale.

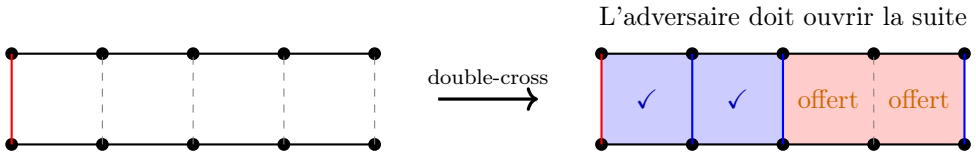
— `MCTSTranspositionTable` : stocke les statistiques de visites et de victoires accumulées par les simulations. Elle sert à initialiser à chaud les nouveaux noeuds MCTS avec des statistiques de tours précédents, et fournit à Alpha-Bêta un ordre de priorité sur les coups à explorer. Les statistiques sont accumulées par addition sur la même position, et la table fait l'objet d'une décroissance périodique entre les tours pour réduire le poids des données obsolètes.

Grâce au hachage de Zobrist partagé, le meilleur coup identifié par Alpha-Bêta est directement exploitable par le MCTS pour orienter son expansion, et réciproquement les statistiques MCTS guident l'ordre d'exploration d'Alpha-Bêta. Cette synergie constitue un des points forts de l'architecture hybride.

5.3.5 Fonction d'évaluation et théorie de la parité

Dans la phase finale d'une partie de Dots and Boxes, une fois les coups *sûrs* épuisés, la partie se réduit à une cascade d'ouvertures de chaînes. Une *chaîne* est une séquence maximale de cases non capturées, chacune ne possédant exactement que deux segments tracés, reliées entre elles par des côtés ouverts. Selon leur longueur, on distingue les *courtes chaînes* ($L \leq 2$), qui peuvent être cédées librement à l'adversaire sans avantage paritatif, et les *longues chaînes* ($L \geq 3$), qui constituent l'enjeu central de la fin de partie.

L'outil stratégique fondamental est le *double-cross* : face à une longue chaîne de longueur L ouverte par l'adversaire, le joueur en position de capturer n'est pas contraint de tout prendre. Il peut capturer $L-2$ cases, puis céder volontairement les 2 dernières, forçant ainsi l'adversaire à ouvrir la chaîne suivante. Ce mécanisme est illustré ci-dessous.



Chaîne longue ouverte par l'adversaire ($L = 4$) — Prend $L - 2 = 2$, offre 2

La question stratégique se réduit donc à : qui sera contraint d'ouvrir la première longue chaîne ? La réponse est déterminée par la *parité* des coups neutres encore disponibles.

On appelle *coup sûr* un coup qui ne ferme pas de case et ne crée pas de case à trois côtés, c'est-à-dire un coup qui n'affecte que des cases comportant au plus un côté tracé. On note n_s le nombre de coups sûrs restants.

On note également n_{sc} le nombre de courtes chaînes dont toutes les cases possèdent encore exactement deux segments, que l'on peut interpréter comme des jetons de parité purs. Le joueur courant détient la parité ssi :

$$(n_s + n_{sc}) \bmod 2 = 1$$

Les longues chaînes n'interviennent pas dans cette formule : une fois la parité acquise, le vainqueur enchaîne les double-cross sur toutes les longues chaînes, quel que soit leur nombre. Leur rôle se limite au gain matériel, calculé comme le gain net par double-cross : $L-2$ cases récupérées pour une longue chaîne ouverte de longueur L, et $L-4$ pour un cycle fermé.

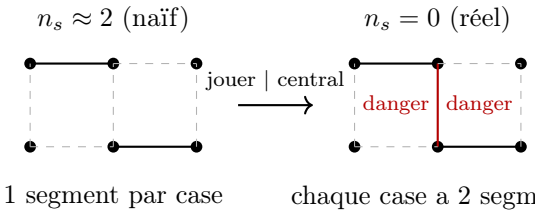
Notre fonction d'évaluation opère selon deux modes distincts selon l'avancement de la partie.

Mode résolu Lorsque $n_s = 0$ et $n_{sc} = 0$, la partie est une pure cascade de captures et le résultat est déterministe. Si le joueur courant dispose d'une capture active dans une longue chaîne, il détient l'initiative et gagne ; sinon, il est contraint d'ouvrir la prochaine chaîne et perd. La valeur retournée est proche de ± 1, modulée par f_1 (différence de score courante) pour ordonner les coups à l'intérieur d'une position gagnante. Les deux termes f_1 et materialAdv sont normalisés par le nombre total de cases du plateau, de sorte que la valeur reste dans $[-1, 1]$ indépendamment de la taille du plateau :

$$v = \tanh(2{,}5 \cdot (f_1 + \text{winSign} \cdot \text{materialAdv})), \quad \text{winSign} \in \{-1, +1\}$$

Mode parité normal Lorsque des coups neutres subsistent, la fonction *estime* quel joueur détient l'avantage de parité et oriente le signal en sa faveur. Ce signal pondère la valeur matérielle attendue des longues chaînes et des courtes chaînes.

Le terme *estimé* est délibéré : le nombre de coups sûrs réels est difficile à déterminer statiquement, car il dépend de l'ordre dans lequel les segments seront joués. Considérons l'exemple suivant, avec deux cases disposées côte à côte, chacune n'ayant qu'un segment tracé :



Un comptage naïf attribue 2 coups sûrs (1 par case). Mais après que l'adversaire joue le segment central, chaque case possède désormais deux segments tracés : tout coup supplémentaire sur l'une d'elles crée immédiatement une case à trois côtés, ce qui est non sûr. Le nombre réel de coups sûrs chute à 0, faisant basculer complètement la parité. Ce phénomène rend le comptage statique peu fiable lorsque de nombreuses cases incomplètes sont encore ouvertes.

C'est pourquoi ce mode de parité n'est activé que lorsque le nombre de cases à zéro ou un segment est faible. À ce stade, la structure du plateau est suffisamment cristallisée pour que l'estimation de n_s soit fiable et que le signal de parité guide correctement la recherche.

5.3.6 Point d'entrée de la décision : `selectAction`

`selectAction` est appelée à chaque tour par l'arbitre. Elle traduit d'abord le plateau vers une représentation *Bitboard* interne (`FastBoard`) pour maximiser la vitesse de simulation, puis orchestre la recherche selon la phase de jeu : Alpha-Bêta exhaustif en fin de partie, MCTS avec assistance tactique Alpha-Bêta en ouverture et milieu de partie. Le coup retenu est enfin décodé vers le format `Action` attendu par l'arbitre.

```
fonction selectAction(board, playerId) :
    fastBoard <- translate(board) // Conversion vers la representation bitboard
    phase <- detectPhase(fastBoard) // Detection de la phase de jeu

    selon phase :
        SMALL_AB, ENDGAME :
            bestEncoded <- abSearch.search(fastBoard, playerId, maxDepth)

        OPENING, MIDGAME :
            // Priorite tactique : saisir toute case immediatement disponible
            pour chaque move dans fastBoard.getAvailableActions() :
                si fastBoard.wouldCloseABox(move) :
                    retourner fastBoard.decode(move)

            // Phase MIDGAME : lancer un AB peu profond avant MCTS
            // pour identifier des sequences de 3-4 coups
            // et alimenter la table de transposition (cross-pollination)
            si phase = MIDGAME :
                abSearch.search(fastBoard, playerId, ABHelperDepth)

            bestEncoded <- mctsSearch.search(fastBoard, playerId, mctsIterations)

    retourner fastBoard.decode(bestEncoded)
```

5.3.7 Bilan

L'architecture de la stratégie *Expert* repose sur une attention particulière portée à l'efficacité computationnelle. Les choix d'implémentation visent à limiter les surcoûts, notamment grâce à une représentation en *Bitboard*, des manipulations bit-à-bit, l'absence d'allocations dynamiques pendant la recherche (à l'exception des nœuds MCTS), ainsi que la réutilisation de structures préallouées. Cette approche s'appuie également sur des principes théoriques du jeu, tels que la parité des chaînes, la détection des boucles et la gestion du double-croisement.

En pratique, ces optimisations permettent à l'algorithme Alpha-Bêta d'explorer des profondeurs plus importantes qu'une implémentation naïve reposant sur des copies complètes du plateau. En fin de partie, lorsque les tables de transposition sont suffisamment renseignées et que l'espace de recherche se réduit, l'agent est en mesure de résoudre certaines configurations par exploration plus profonde. Du côté du MCTS, la représentation par bitboard et la réduction des allocations permettent d'augmenter le nombre de simulations réalisables dans un même budget de calcul.

Évaluation expérimentale Nous avons évalué la stratégie `ExpertActionStrategy` contre trois stratégies de référence sur une grille de 4×4 points. Chaque confrontation correspond à 80 parties au total, afin de tester les deux ordres de jeu.

Confrontation	Victoires Expert	Défaites Expert	Nuls	Score Expert	Score adverse
Expert vs Glouton	80	0	0	440	280
Expert vs Minimax	73	7	0	575	145
Expert vs Alpha-Beta	70	10	0	476	244

Ces résultats montrent un gain mesurable par rapport aux stratégies précédentes. L'agent expert gagne toutes les parties contre la stratégie gloutonne, puis conserve un avantage net contre Minimax et Alpha-Beta. Cela confirme l'intérêt de l'approche hybride : combiner recherche, heuristiques de chaînes, bitboard et tables de transposition améliore la qualité pratique des décisions.

Des améliorations restent toutefois possibles. En particulier, la théorie de la parité n'est pas encore pleinement maîtrisée dans notre implémentation. Des tests empiriques ont mis en évidence certaines faiblesses en fin de partie, notamment face à des agents de très haut niveau, où des décisions sous-optimales peuvent survenir lorsque les mécanismes liés à la parité ne sont pas encore pleinement pris en compte.

Par ailleurs, il serait pertinent d'affiner la fonction d'évaluation à l'aide de techniques d'apprentissage automatique, afin d'ajuster automatiquement les poids heuristiques.

6 Tests et validation

Afin de garantir la fiabilité de l'implémentation, un ensemble complet de tests unitaires a été développé à l'aide du framework JUnit. La suite de tests actuelle contient 75 tests, tous validés lors de l'exécution complète avec Maven.

Les tests portant sur la classe `Board` vérifient les opérations fondamentales : initialisation du plateau, validation des actions, fermeture d'une ou deux cases, mise à jour des scores, fin de partie et copie profonde des structures de données. Les tests de `DotsBoxesGame` et `Referee` vérifient la boucle de jeu, la gestion des coups invalides, le calcul du gagnant et la règle du tour supplémentaire.

Les stratégies font également l'objet de tests ciblés. `GloutonActionStrategy` est testée sur la fermeture immédiate d'une case et sur l'évitement d'un coup qui donnerait une capture immédiate à l'adversaire. `MinimaxActionStrategy` est testée sur les états terminaux, les fermetures de cases, le maintien du même joueur après capture, la fermeture de deux cases et l'évitement des cases dangereuses. `AlphaBetaActionStrategy` est comparée à Minimax sur des positions représentatives afin de vérifier la cohérence des décisions. Enfin, `ExpertActionStrategy` est testée sur la validité des coups retournés, la fermeture immédiate, l'évitement des positions dangereuses et le comportement avec différentes profondeurs de recherche.

Des tests d'expérimentation complètent ces tests unitaires. Ils permettent d'exécuter les comparaisons Random/Glouton, Minimax/Alpha-Beta, l'analyse de profondeur de Minimax et l'évaluation de la stratégie experte contre les stratégies de référence.

7 Conclusion

Ce projet a permis de modéliser le jeu *Dots and Boxes* comme un jeu déterministe à deux joueurs, à information parfaite, puis d'implémenter plusieurs stratégies de complexité croissante. La représentation du plateau par segments horizontaux, segments verticaux et propriétaires des cases fournit une base simple pour appliquer les règles du jeu, détecter les captures et gérer le tour supplémentaire.

Les expérimentations montrent d'abord qu'une stratégie gloutonne est nettement plus efficace qu'une stratégie aléatoire, mais qu'elle reste limitée dans les configurations de chaînes. L'introduction de Minimax permet d'anticiper les réponses adverses, au prix d'une croissance rapide du temps de calcul lorsque la profondeur augmente. Alpha-Beta réduit fortement ce coût grâce à l'élagage, en particulier sur les plateaux plus grands et aux profondeurs plus élevées.

Enfin, la stratégie *Expert* améliore les approches précédentes en combinant une représentation par bitboard, des tables de transposition, une recherche Alpha-Bêta plus efficace, du MCTS et des heuristiques liées aux chaînes et à la parité. Les résultats expérimentaux montrent un avantage net contre Glouton, Minimax et Alpha-Beta sur la configuration testée, ce qui valide l'intérêt de cette approche hybride.

Des améliorations restent possibles, notamment un traitement plus robuste de certaines situations de fin de partie, une meilleure maîtrise de la parité des chaînes et l'ajustement automatique des poids heuristiques.