SIM202

Projet d'informatique en C++:

Implémenter une Intelligence Artificielle jouant de manière optimale aux échecs

Felix Azian & Jeremy Borrega & Abdoulaye Traore

Mars 2025

Contents

1 Introduction

Ce projet a pour objectif de découvrir et d'implémenter des algorithmes classiques utilisés en théorie des jeux pour des jeux à deux joueurs et à information complète. Ces algorithmes seront appliqués au jeu d'échec qui nécessite de prendre en compte énormément d'informations étant donné la complexité du jeu, la multitude de coups possibles à jouer, et des coups spéciaux (notamment le roque et le "en passant").

Les deux algorithmes implémentés dans ce projet sont **Minimax** et **Alpha-Beta**. Pour traiter ce projet, nous avons eu à faire des choix au niveau de la structure du code et des différentes classes introduites pour améliorer la complexité de notre algorithme qui peut rapidement devenir conséquente, et nous avons rencontré plusieurs difficultés pour parvenir à concevoir le code final.

Ce rapport a donc pour objectif d'expliquer nos difficultés et les idées que nous avons eu au cours de ce projet pour essayer d'y remédier, mais également de montrer comment nous avons validé nos résultats.

2 Répartition du travail

Le travail a été répartie entre nous afin d'assurer une avancée fluide du projet.

1. **Mise en place de Tic-Tac-Toe et du jeu d'échecs :** Dans un premier temps, l'objectif était de mettre en place, sans considérer une IA pour le moment, les jeux Tic-Tac-Toe et le jeu d'échecs qui seront les deux jeux sur lesquels nous allons travailler.

   Jeremy a entrepris l'implémentation de toutes les règles de jeu pour le jeu d'Echec et de TicTacToe.

2. **Intégration d'une IA et des algorithmes de jeu :** Nous devions ensuite mettre en place l'algorithme Minimax et toutes les fonctions nécessaires à son implémentation. Cela reposait sur la mise en place de la classe abstraite `Position`, qui constitue la partie centrale de ce projet, car elle est héritée par les classes `PositionTicTacToe` et `PositionEchec`.

   Felix a été responsable de la création de la classe `Position` et de l'implémentation de l'algorithme Minimax générique, applicable tant au Tic-Tac-Toe qu'au jeu d'échecs, en développant également la création d'arbres de positions et des méthodes de calcul heuristique pour attribuer une valeur à chaque coup joué et déterminer le coup le plus optimal à jouer.

   Une fois la classe `Position` définie, le travail a été divisé de la manière suivante :

   - **Felix** a défini les méthodes virtuelles pures et a pris en charge le développement de la classe héritée de Position, `PositionTicTacToe`, ainsi que des classes sous-jacentes pour `TicTacToe`.
   - **Abdoulaye** s'est chargé de définir et implémenter toutes les méthodes de la classe dérivée `PositionEchec` à savoir la création et le chaînage des positions filles, la définition des heuristiques et les méthodes de création et d'annulation de coups.

3. **Améliorations et optimisations :** Une fois l'implémentation de base terminée, le groupe s'est concentré sur des améliorations importantes pour optimiser les performances et enrichir l'expérience de jeu.

   - **Abdoulaye** a pris en charge l'implémentation de l'élagage Alpha-Beta pour améliorer l'efficacité de l'algorithme Minimax, réduisant ainsi le nombre de nœuds explorés ainsi que la mise en place des tests et de coups spéciaux.
   - **Jeremy** a ajouté la gestion des ouvertures dans le jeu d'échecs, permettant de jouer avec des ouvertures classiques et prendre le contrôle de l'échiquier.
   - **Felix** a pris en charge toute la documentation du projet incluant le README et le rapport du projet.

3 Architecture du code

Notre dossier est structuré de la manière suivante :

```
/projet-echecs
 src/
    main.cpp        # Programme principal
    echiquier.hpp   # Déclaration de la classe Echiquier
    echiquier.cpp   # Implémentation de la classe Echiquier
    piece.hpp       # Déclaration de la classe Piece
    piece.cpp       # Implémentation de la classe Piece
    coup.hpp        # Déclaration de la classe Coup
    coup.cpp        # Implémentation de la classe Coup
    openings.cpp        # Implémentation des ouvertures
    openings.hpp        #  Déclaration de la classe Openings
    Test.cpp        #  Implémentation de tous les tests
    Test.hpp        #  Déclaration des différents tests
    Makefile        #  Automatisation de la compilation
    README.md        #  Instructions pour la compilation
```

Voici une description brève des fichiers :

- **main.cpp** : Contient le programme principal qui initialise le jeu et gère son exécution.
- **openings.cpp** : Contient l'implémentation des ouvertures d'échecs. Ce fichier définit les mouvements d'ouverture utilisés dans le jeu, permettant d'initier la partie avec des configurations reconnues.
- **openings.hpp** : Déclare la classe `Openings` qui gère les différentes ouvertures dans le jeu d'échecs.
- **Test.cpp** : Contient l'implémentation de tous les tests. Ce fichier est responsable de l'exécution des tests pour vérifier le bon fonctionnement des classes et méthodes du projet.
- **Test.hpp** : Déclare les différents tests.
- **echiquier.hpp** : Définit la classe `Echiquier`, qui représente le plateau de jeu. Elle contient les déclarations des méthodes permettant la gestion des pièces et du plateau.
- **echiquier.cpp** : Implémente les méthodes de la classe `Echiquier`. Ce fichier gère l'affichage de l'échiquier, le déplacement des pièces et la détection des coups légaux.
- **piece.hpp** : Déclare la classe `Piece`, qui représente une pièce du jeu d'échecs. Elle définit les attributs essentiels comme le type de la pièce, sa couleur et sa position.
- **piece.cpp** : Implémente les comportements des différentes pièces du jeu. Il contient les méthodes permettant d'obtenir les informations des pièces et de modifier leur état, comme leur position sur l'échiquier.
- **coup.hpp** : Déclare la classe `Coup`, qui représente un mouvement effectué par une pièce. Cette classe stocke les informations du déplacement, telles que la position initiale et finale, la pièce impliquée, et les éventuelles captures.
- **coup.cpp** : Implémente les méthodes de la classe `Coup`. Ce fichier gère l'application des mouvements sur l'échiquier, en vérifiant leur validité et en mettant à jour les positions des pièces après chaque coup.
- **README.md** : Documentation du projet, expliquant les fonctionnalités, la structure, et les instructions pour jouer, ainsi que les tests disponibles via le fichier `main.cpp`.

4 Implémentation de l'algorithme Minimax

Étant donné la complexité du jeu d'échecs, nous avons d'abord testé l'implémentation de l'algorithme Minimax sur un jeu plus simple : le Tic-Tac-Toe. Ce choix nous a permis de valider notre compréhension de l'algorithme dans un environnement où **l'espace des états est réduit et où toutes les configurations possibles peuvent être facilement explorées**. Le **Tic-Tac-Toe** est un jeu déterministe avec un nombre limité de coups possibles, ce qui nous a permis de visualiser l'arbre de décision complet et de vérifier l'efficacité de Minimax sans avoir à gérer des heuristiques complexes.

Une fois l'algorithme fonctionnel et optimisé pour le Tic-Tac-Toe, nous avons progressivement transposé son implémentation aux échecs. **La principale difficulté réside dans la taille exponentielle de l'espace des états aux échecs**, nécessitant l'intégration d'élagage alpha-bêta pour réduire le nombre de nœuds explorés. Ce dernier sera la sujet d'une partie suivante.

4.1 Idée générale de l'algorithme Minimax

L'algorithme Minimax est un algorithme de recherche utilisé dans les jeux à somme nulle, où deux joueurs s'affrontent avec l'objectif de **maximiser leur propre score tout en minimisant celui de l'adversaire**.

L'algorithme Minimax fonctionne alors selon deux principes fondamentaux :

- **Maximiser :** Le joueur qui maximise cherche à obtenir la valeur la plus élevée pour sa position.
- **Minimiser :** Le joueur qui minimise cherche à obtenir la valeur la plus basse pour sa position.

À chaque étape, l'algorithme explore les coups possibles et, pour chaque coup, applique récursivement la même procédure pour évaluer les conséquences. Cette exploration des coups possibles continue jusqu'à ce que l'un des critères d'arrêt soit atteint :

- Un gagnant a été déterminé.
- La profondeur maximale a été atteinte.
- Aucun coup valide n'est disponible (match nul).

L'algorithme calcule une valeur pour chaque position, et à chaque niveau de la recherche, le joueur choisissant un coup doit décider si le coup doit maximiser ou minimiser la valeur. Pour ce faire, il compare les valeurs des positions obtenues à travers les coups et sélectionne la valeur maximale ou minimale en fonction du joueur.

Pseudo-code de l'algorithme Minimax

```
Fonction minimax(Plateau Board_ref) : flot

    Initialiser l'échiquier avec MakeEchiquier(Board_ref)

    gagnant = Board_ref.verifierGagnant()   // Vérifier s'il y a un gagnant immédiat

    Si (gagnant == 'O' ou gagnant == 'N') :   // Victoire
        Retourner INFINITY

    Si (gagnant == 'X' ou gagnant == 'B') :   // Défaite
        Retourner -INFINITY

    Si (gagnant == 'V') :  // Egalité
```

```
    Retourner 0

Si (depth >= profondeur_max) :   // Condition d'arrêt : profondeur maximale atteinte
    Retourner heuristique(Board_ref)

Si (Aucun coup possible) :
    Retourner heuristique(Board_ref)  // Match nul

BestVal = -INFINITY Si maximise == True Sinon INFINITY  // Initialiser la meilleure valeur

Pour chaque position enfant dans les positions filles générées :
    valeur = enfant.minimax(Board_ref)
    Si maximise == True :
        BestVal = max(BestVal, valeur)
    Sinon :
        BestVal = min(BestVal, valeur)

Retourner BestVal
```

En résumé, l'algorithme Minimax permet de prendre une décision optimale en simulant toutes les possibilités de coups futurs dans un jeu, en supposant que l'adversaire joue de manière optimale également.

4.2 Structure de la classe Position

Nous allons dans cette partie expliquer la structure de la classe Position qui est centrale pour mettre en place nos algorithmes de recherche.

La classe abstraite `Position` définit la structure de base pour modéliser les différentes positions d'un jeu dans le cadre de l'algorithme Minimax. Elle contient des pointeurs vers les positions filles et soeurs, ainsi que des attributs comme la profondeur et l'indicateur de joueur. Les méthodes clés incluent `MakeEchiquier`, `UndoEchiquier`, `heuristique` et `genererFilles`, qui sont spécifiques aux sous-classes.

Les classes dérivées, `position_tictactoe` et `position_echec`, héritent de `Position` et implémentent ces méthodes pour les jeux spécifiques. `position_tictactoe` est adaptée au jeu du Tic-Tac-Toe, tandis que `position_echec` est destinée aux échecs, avec des règles et évaluations propres à chaque jeu. Cette structure permet de réutiliser l'algorithme Minimax dans différents contextes de jeux.

4.3 Problèmes rencontrés

Nous avons constaté lors de l'implémentation de l'algorithme Minimax que l'IA ne joue pas de manière optimale. Par exemple, lorsqu'il reste des cases libres, l'IA ne joue pas au centre bien que cela soit la position la plus stratégique dans le Tic-Tac-Toe.
Nous avons donc entrepris un travail de débogage pour identifier la source du problème. Nous avons d'abord envisagé que notre calcul d'heuristique soit erroné et pensé qu'il serait nécessaire d'augmenter l'heuristique si l'IA joue au centre. Cependant, le problème résidait ailleurs.
En effet, si les positions filles étaient générées correctement, il n'y aurait pas besoin d'augmenter l'heuristique pour la position centrale. Lors du parcours de l'arbre, l'IA devrait déjà identifier que jouer au centre mène à plus de possibilités de victoire. Nous nous sommes donc concentrés sur la fonction `genererFilles` et avons constaté que les positions filles ne sont pas générées correctement.
Après avoir réglé les problèmes apparents, il faut comprendre si notre algorithme fonctionne vraiment dans toutes les situations de jeu.

4.4 Validation des résultats de l'IA avec Minimax dans le jeu Tic-Tac-Toe

Dans cette section, nous expliquons comment nous avons validé les résultats obtenus lors de l'implémentation de l'algorithme Minimax pour un jeu Tic-Tac-Toe et Échecs. Pour ce faire, nous avons utilisé le fichier `Test.cpp` afin d'effectuer plusieurs tests unitaires et de centraliser les différents tests.
Comme nous allons par la suite tester l'élagage Alpha-Beta, nous avons introduit un booléen `estAlpha` qui vaut false si on veut utiliser Minimax et true si on veut utiliser Alpha-Beta. Ce booléen apparaît notamment dans l'algorithme `meilleurCoup` étant donné que le meilleur coup n'est pas détérminé de la même manière en fonction de l'algorithme de jeu utilisé.

Tests pour le Tic-Tac-Toe

Nous avons commencé par **tester l'IA Minimax contre un joueur humain** dans un environnement de jeu Tic-Tac-Toe. Le test a consisté à valider si l'IA joue de manière optimale et si elle respecte les règles du jeu. Les tests effectués incluent :

- **Coup Invalide :** Tentative de jouer sur une case déjà occupée.
- **Coup Gagnant:** L'IA doit jouer le coup gagnant lorsque la situation est favorable, comme remplir la dernière case d'une ligne pour gagner.
- **Match Nul :** Remplir le plateau sans qu'aucun joueur ne gagne.

Ensuite nous avons voulu automatiser cela en faisant jouer 50 parties à l'IA contre un joueur jouant de manière aléatoire pour s'assurer que l'algorithme fonctionne dans toute situation.

Un autre test crucial a consisté à **faire jouer l'IA contre elle-même** pour observer si l'algorithme Alpha-Beta optimisait effectivement ses coups. Le nombre de victoires et de matchs nuls a été suivi pour évaluer la performance globale de l'IA. Intuitivement, si les deux jouent de manière optimale, on doit toujours faire match nul, et c'est bien ce qu'on observe.

Figure 1: Résultats sur 50 parties: IA Minimax vs Random

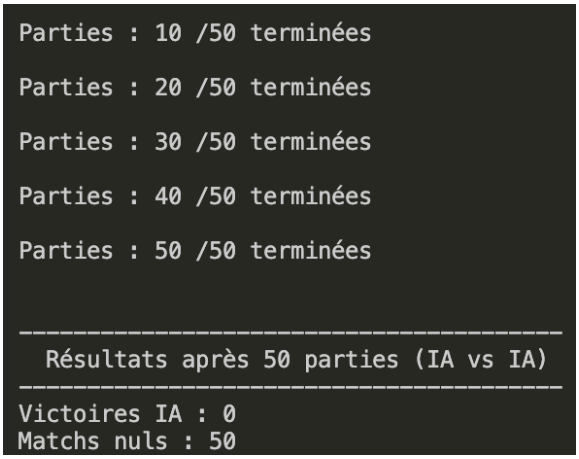

Figure 2: Résultats sur 50 parties: IA Minimax vs IA Minimax

4.5 Adaptation de l'algorithme Minimax pour les Échecs

L'algorithme Minimax reste fondamentalement le même, mais la manière dont on va générer les positions filles et attribuer une heuristique pour chaque position a été adapté pour le jeu d'échecs afin d'intégrer des mécanismes spécifiques à ce jeu complexe.

Génération des Positions Filles

L'algorithme commence par générer les positions filles à partir de l'état actuel de l'échiquier. Pour cela, chaque pièce est analysée pour identifier tous les **coups possibles** qu'elle peut réaliser. Ce processus est spécifique à chaque type de pièce (roi, reine, cavalier, etc.), et la génération des coups prend en compte les règles de déplacement et les captures. L'implémentation de la fonction `genererFilles` permet de parcourir chaque pièce du joueur actif et de générer ses coups valides.

Heuristique pour les Échecs

L'heuristique utilisée dans le cadre des échecs prend en compte plusieurs facteurs afin d'évaluer la qualité d'une position donnée. Elle repose sur les critères suivants :

- **Valeur des pièces** : Chaque type de pièce a une valeur attribuée, par exemple, la reine est plus puissante que le cavalier. L'heuristique additionne ces valeurs pour les pièces du joueur et de l'adversaire.
- **Contrôle du centre** : Le contrôle des cases centrales (comme les cases d4, d5, e4, e5) est important car il permet une plus grande mobilité des pièces. L'heuristique favorise les positions où les pièces contrôlent ces cases centrales.
- **Mobilité des pièces** : Plus une pièce a de possibilités de déplacement, plus elle est puissante. Ainsi, l'heuristique considère la mobilité globale des pièces pour évaluer une position.
- **Sécurité du roi** : L'heuristique pénalise les positions où le roi est exposé à des attaques directes ou indirectes (en échec).
- **Équilibre matériel** : Un joueur avec plus de pièces ou de matériel (pions, tours, cavaliers) est généralement dans une position favorable.

L'heuristique utilisée pour évaluer une position dans un jeu d'échecs repose ainsi sur plusieurs critères. Ces critères sont combinés dans une formule pondérée, où chaque facteur a un poids associé en fonction de son importance dans la position actuelle. La formule générale de l'heuristique est la suivante :

$$H = \alpha \cdot (V_{\text{IA}} - V_{\text{Adversaire}}) + \beta \cdot (C_{\text{IA}} - C_{\text{Adversaire}}) + \gamma \cdot (M_{\text{IA}} - M_{\text{Adversaire}}) + \delta \cdot (C_{\text{centre, IA}} - C_{\text{centre, Adversaire}})$$

où :

- V_{IA} et $V_{\text{Adversaire}}$ représentent la valeur totale des pièces respectivement pour l'IA et pour l'adversaire.
- C_{IA} et $C_{\text{Adversaire}}$ représentent le contrôle des cases centrales pour l'IA et pour l'adversaire, en fonction du nombre de pièces qui contrôlent ces cases.
- M_{IA} et $M_{\text{Adversaire}}$ représentent la mobilité des pièces pour l'IA et pour l'adversaire, c'est-à-dire le nombre de déplacements possibles pour les pièces respectives.
- $C_{\text{centre, IA}}$ et $C_{\text{centre, Adversaire}}$ représentent le contrôle des cases centrales (par exemple, les cases d4, e4, d5, e5) respectivement pour l'IA et pour l'adversaire.
- Les coefficients α, β, γ et δ sont des poids qui reflètent l'importance relative de chaque critère. Par exemple, α pourrait être plus grand si la valeur des pièces est plus critique que la mobilité des pièces, tandis que δ pourrait être plus élevé pour privilégier le contrôle du centre.

5 Passage de Minimax à Alpha-Beta

5.1 Motivations

L'algorithme Minimax explore récursivement les positions possibles jusqu'à une **profondeur maximale** (nécessaire d'introduire étant donné le nombre de coups possibles dans un jeu d'échecs), où il évalue la position à l'aide de l'heuristique. Pour optimiser cette recherche, l'algorithme Alpha-Beta est utilisé pour élaguer les branches non pertinentes de l'arbre de décision, réduisant ainsi le nombre de positions à explorer.

5.2 Fonctionnement global de l'algorithme Alpha-Beta

L'algorithme alpha-bêta est une amélioration de l'algorithme Minimax qui permet de réduire le nombre de nœuds explorés dans l'arbre de décision. Cet algorithme utilise deux valeurs, **alpha** et **beta**, pour effectuer des *coupes* qui éliminent certaines branches de l'arbre de recherche. Ces coupes permettent d'éviter de parcourir des positions inutiles, ce qui rend l'algorithme plus efficace.

Coupe Alpha

La coupe alpha se produit lorsque l'algorithme découvre qu'un joueur (l'un des deux adversaires) ne peut pas améliorer son score au-delà d'une valeur donnée. Plus précisément, si l'algorithme minimise le score (recherche d'une position de plus faible valeur), et que ce score est plus bas que **alpha**, alors les positions suivantes n'ont aucune chance de surpasser cette valeur et sont donc ignorées.

$$\text{Si valeur} \leq \alpha \quad \text{alors on effectue une coupe alpha.}$$

Cette coupe permet de *couper* l'exploration de l'arbre lorsque le score trouvé est déjà inférieur à la meilleure valeur déjà trouvée pour le joueur maximisant. Cela signifie que le joueur minimisant ne va pas pouvoir obtenir une valeur plus favorable dans cette branche, et il est inutile de l'explorer davantage.

Coupe Beta

La coupe beta, en revanche, se produit lorsque l'algorithme découvre qu'un joueur (le joueur maximisant) ne pourra pas obtenir un score supérieur à une valeur donnée. Si le score du joueur maximisant est plus grand que **beta**, on effectue une coupe, car cela signifie que le joueur maximisant ne pourra pas faire un meilleur coup que celui déjà trouvé.

$$\text{Si valeur} \geq \beta \quad \text{alors on effectue une coupe beta.}$$

Cette coupe permet de *couper* l'exploration de l'arbre lorsqu'une valeur plus grande que celle déjà trouvée pour le joueur minimisant est trouvée. Cela permet de gagner du temps en évitant de parcourir des branches qui ne peuvent pas offrir un meilleur score.

Pseudo-Code

```
Fonction AlphaBeta(Plateau Board_ref, float alpha, float beta, char symboleIA):
    // Évaluer la position actuelle
    FaireEchiquier(Board_ref)
    gagnant = Board_ref.verifierGagnant()
    AnnulerEchiquier(Board_ref)

    // Vérifier si l'IA a gagné
    Si gagnant == symboleIA:
        Retourner INFINITY  // L'IA a gagné
```

```
// Vérifier si l'humain a gagné
SymboleH = Board_ref.getSymboleH(symboleIA)  // Récupérer le symbole de l'humain
Si gagnant == SymboleH:
    Retourner -INFINITY  // L'humain a gagné

// Vérifier si c'est un match nul
Si gagnant == 'V':
    Retourner 0  // Match nul

// Si la profondeur maximale est atteinte, utiliser l'heuristique
Si depth >= getDepthMax(Board_ref.getTypePlateau()):
    Retourner heuristique(Board_ref, symboleIA)

// Générer les positions enfants
Si !genererFilles(Board_ref, symboleIA):
    Retourner heuristique(Board_ref, symboleIA)

// Initialiser la meilleure valeur en fonction du joueur (maximiser ou minimiser)
Si maximise:
    bestValue = -INFINITY
Sinon:
    bestValue = INFINITY

// Parcourir tous les enfants générés
Pour chaque enfant dans les filles:
    // Appeler récursivement alphaBeta pour obtenir la valeur de l'enfant
    valeur = enfant.alphaBeta(Board_ref, alpha, beta, symboleIA)

    Si maximise:  // Si c'est le tour du joueur maximisant (l'IA)
        Si valeur > bestValue:
            bestValue = valeur
            alpha = max(alpha, valeur)  // Mettre à jour alpha

        Si valeur >= beta:  // Coupure beta
            Retourner valeur  // On a trouvé une meilleure branche

    Sinon:  // Si c'est le tour du joueur minimisant (l'humain)
        Si valeur < bestValue:
            bestValue = valeur
            beta = min(beta, valeur)  // Mettre à jour beta

        Si valeur <= alpha:  // Coupure alpha
            Retourner valeur  // On a trouvé une meilleure branche

// Retourner la meilleure valeur trouvée
Retourner bestValue
```

5.3 Validation des algorithmes Minimax et Alpha Beta sur le jeu d'échec

5.3.1 Validation préliminaire de l'élagage Alpha-Bêta sur Tic-Tac-Toe

De la même manière qu'avec l'algorithme Minimax, nous avons validé notre algorithme Alpha-Beta sur le Tic-Tac-Toe et avons réalisé un test qui consiste à faire jouer l'IA contre un joueur jouant de manière aléatoire.
Les résultats obtenus sont satisfaisants, mais ce qui est surtout satisfaisant c'est la rapidité avec laquelle l'algorithme détermine le coup le plus optimal. On voit une vraie amélioration au niveau de la complexité de l'algorithme sur un jeu aussi simple que le Tic-Tac-Toe ce qui nous amène à penser que cette efficacité se reflétera sur le jeu d'échecs.

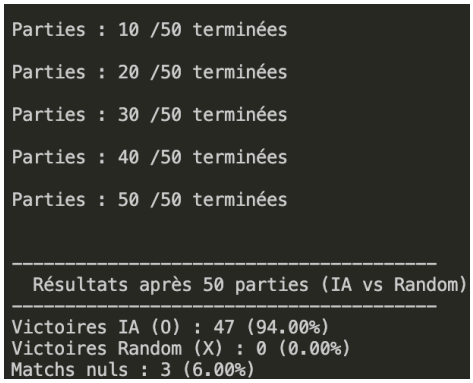

Figure 3: Résultats sur 50 parties: IA Alpha-Beta vs Random

5.3.2 Idée pour valider les algorithmes sur le jeu d'échecs

Le jeu d'échecs étant plus complexe que le Tic-Tac-Toe, il est difficile de réaliser autant de parties simulées. Nous avons plutôt focalisé notre attention sur des cas spécifiques et des tests de situations particulières comme les *roques*, les *prises en passant*, la *promotion de pion*, et la détection de situations de *stalemate* et *checkmate*.

Les tests sont réalisés dans les conditions suivantes :

- Le `Petit Roque Valide` et le `Grand Roque Valide` sont testés pour vérifier que l'IA peut effectuer ces mouvements spéciaux en respectant les règles du jeu.
- Le test `Roque Invalide` s'assure que l'IA est capable de détecter si le chemin du roi est bloqué et de rejeter le mouvement.
- Le test `Prise en Passant Valide` vérifie si l'IA peut appliquer correctement cette règle spéciale dans des situations de capture.
- Le test de la `Promotion de Pion` vérifie que l'IA peut promouvoir un pion lorsqu'il atteint la dernière rangée.
- Le `Mouvement du Cavalier` vérifie que l'IA prend en compte les déplacements possibles du cavalier, qui a un mode de déplacement unique.
- Les tests de `Stalemate` (match nul) et de `Checkmate` (mat) vérifient que l'IA détecte correctement ces conditions de fin de partie.

Pour chacun de ces tests, l'IA doit détecter les bonnes stratégies et appliquer les règles du jeu de manière correcte. Par exemple, pour les roques, l'IA doit s'assurer que les cases entre le roi et la tour sont libres et que le roi n'est pas en échec avant d'effectuer le mouvement.

Nous avons utilisé les fonctions définies dans `test.cpp` pour valider chaque fonctionnalité spécifique de l'IA dans des situations réelles de jeu.

6 Amélioration de l'IA

6.1 Ouvertures classiques et gestion de la transition

Pour améliorer la stratégie de notre IA dans le jeu d'échecs, nous avons intégré un module permettant à l'IA de jouer des **ouvertures classiques**. Ces ouvertures sont des séquences de coups préétablies utilisées par les meilleurs joueurs, visant à développer les pièces rapidement et à sécuriser le roi dès le début de la partie. Leur intégration permet d'offrir à l'IA un jeu plus naturel et cohérent, tout en minimisant les erreurs stratégiques en début de partie.

6.1.1 Intégration des ouvertures

Dans le fichier `openings.cpp`, nous avons implémenté un système permettant à l'IA de sélectionner une ouverture en fonction de l'historique des coups joués. Contrairement à un simple choix d'ouverture fixe, l'IA **peut s'adapter dynamiquement** et **choisir parmi plusieurs ouvertures possibles**, en fonction des décisions prises par le joueur humain.

Le processus d'intégration des ouvertures suit les étapes suivantes :

1. **Chargement des ouvertures** : Une base de données contenant les principales ouvertures (telles que la Ruy López, la Défense Sicilienne, le Gambit Dame, etc.) est initialisée au début de la partie.
2. **Sélection dynamique d'une ouverture** : Lorsqu'il est au tour de l'IA de jouer, elle consulte l'**historique des coups** pour vérifier si une ou plusieurs ouvertures classiques sont encore possibles.
3. **Choix aléatoire parmi les coups possibles** : Si plusieurs ouvertures sont toujours poursuivables en fonction de l'historique, l'IA sélectionne **aléatoirement un coup parmi ces ouvertures**, ajoutant ainsi de la diversité dans son jeu.
4. **Application du coup** : L'IA applique alors le coup sélectionné à partir des ouvertures classiques.

6.1.2 Transition vers le milieu de jeu

L'IA reste en phase d'ouverture tant que les coups du joueur humain **correspondent encore à une ou plusieurs ouvertures classiques**. Dès que le joueur humain joue un coup qui **ne figure plus dans aucune des ouvertures enregistrées**, l'IA considère que la phase d'ouverture est terminée et elle bascule vers une approche plus flexible.

Une fois sortie de l'ouverture, l'IA adopte une stratégie basée sur l'évaluation des positions, en utilisant l'**algorithme Minimax avec élagage Alpha-Beta** pour maximiser ses chances de victoire.

6.1.3 Fonctionnement interne de getOpeningMove()

La fonction `getOpeningMove()` située dans `openings.cpp` est responsable de la gestion des ouvertures. Elle fonctionne comme suit :

1. Elle prend en paramètre **l'historique des coups joués**.
2. Elle compare cet historique avec **toutes les ouvertures stockées dans la base de données**.
3. Elle identifie **toutes les ouvertures encore possibles à ce stade de la partie**.
4. Elle sélectionne **un coup aléatoire parmi ces ouvertures encore viables**.

Si aucune ouverture ne correspond plus à l'historique actuel, l'IA **désactive la phase d'ouverture** et bascule automatiquement vers Minimax.

6.1.4 Impact sur le jeu de l'IA

Grâce à cette approche :

- L'IA joue de manière plus variée et réaliste, **évite de toujours suivre la même ouverture**.
- La transition entre ouverture et milieu de partie est **fluide et naturelle**.
- L'IA n'est pas forcée de suivre une seule ouverture prédéfinie, mais peut **choisir dynamiquement en fonction de l'adversaire**.

Cette gestion intelligente des ouvertures permet d'améliorer le réalisme du jeu et de renforcer les capacités stratégiques de l'IA.

6.1.5 Format de l'historique des coups et exemple

L'historique des coups est un **vecteur de chaînes de caractères** où chaque élément suit le format :

- **Première lettre** : Couleur du joueur ('W' pour Blanc, 'B' pour Noir).
- **Deuxième et troisième caractères** : Case de départ (ex : 'g5').
- **Quatrième et cinquième caractères** : Case d'arrivée (ex : 'e5').

Ainsi, une partie de l'historique peut ressembler à ceci :

```
["Wg4e4", "Ba7c6", "Wg3e3", "Bb3d3", "We4d4"]
```

Cet historique correspond aux trois premiers coups de l'ouverture **Benoni Moderne** définie comme suit :

```
ouvertureMoves["benoni_modern"] = {
    "Wg4e4", "Ba7c6",
    "Wg3e3", "Bb3d3",
    "We4d4", "Bb5c5",
    "Wh2f2", "Bc5d4",
    "We3d4", "Bb4c4"
};
```

6.1.6 Pseudo-code de `getOpeningMove()`

La fonction `getOpeningMove()` prend en paramètre l'**historique des coups** et sélectionne un coup d'ouverture si l'historique correspond encore à une ouverture enregistrée.
Ce pseudo-code montre que l'IA :

- Vérifie si l'historique correspond au début d'une ou plusieurs ouvertures.
- Récupère tous les coups encore possibles à ce stade de la partie.
- Sélectionne un coup aléatoire parmi les choix restants.
- Indique clairement à quelle ouverture elle se réfère.
- Si aucun coup d'ouverture n'est trouvable, elle bascule dans une phase de jeu basée sur Min-Max.

Voici le fonctionnement de l'algorithme sous forme de pseudo-code :

```
Entrée : historique (vecteur de strings contenant les coups joués)
Sortie : coup d'ouverture sous forme de coup ou coup invalide si aucune ouverture ne correspond

Début
    coupsPossibles ← liste vide
    ouvertureChoisie ← ""

    Pour chaque ouverture (nom, liste de coups) dans ouvertureMoves :
        Si historique est un préfixe de la liste de coups :
            Ajouter le coup suivant de cette ouverture à coupsPossibles
            Si première correspondance trouvée, stocker le nom de l'ouverture

    Si coupsPossibles est vide :
        Retourner un coup invalide

    Mélanger coupsPossibles pour obtenir un choix aléatoire
    coupChoisi ← premier élément de coupsPossibles

    Afficher "L'IA joue dans l'ouverture : " + ouvertureChoisie
    Retourner coupChoisi
Fin
```

7 Conclusion

Dans ce projet, nous avons non seulement appris à travailler en groupe sur un projet informatique, mais nous avons également été confrontés à plusieurs difficultés. L'une des leçons les plus importantes que nous avons tirées de cette expérience est l'importance cruciale de la communication au sein de l'équipe. Une communication efficace permet de garantir une avancée fluide et de mieux répartir les tâches, ce qui est indispensable pour mener à bien un projet de cette envergure.

Ce projet nous a permis d'approfondir nos connaissances en gestion de la mémoire, notamment en utilisant des pointeurs intelligents tels que `unique_ptr`. L'utilisation de ces types de pointeurs a été essentielle pour assurer la gestion de la mémoire dynamique de manière sécurisée et éviter les fuites de mémoire.

Enfin, pour réaliser ce projet, nous avons dû comprendre en profondeur le fonctionnement interne de l'ordinateur, en particulier comment la mémoire est allouée lorsqu'on crée un arbre via une liste chaînée. Cela nous a permis de mieux appréhender les implications de l'utilisation des structures de données complexes et d'optimiser la gestion de ces structures.

Ce projet a donc été une excellente opportunité d'apprentissage, tant sur le plan technique que collaboratif, et nous a permis de renforcer nos compétences en C++, en gestion de la mémoire et des structures de classes, ainsi qu'en travail d'équipe.