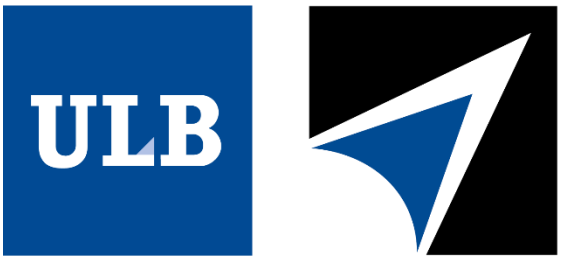



BRUSSELS SCHOOL OF ENGINEERING

# Compléments de Programmation et d'Algorithmique

## INFO-H304

---

## PROJET Groupe n°14

## REVERSI / OTHELLO

---

**Auteur :**

**Esiev Kerim 000493521**

**François Nathan 000514349**

**Sharof Imad 000494736**

**Date : 19/12/2023**

**Professeurs : ROLLAND Jeremy**

Année Académique 2023–2024

# Table des matières

# 1. Introduction

Dans le cadre du cours avancé de Compléments de programmation et d'algorithmique, les élèves de troisième bachelier ont été confrontés à un défi d'appliquer et d'étendre les compétences en programmation et algorithmes à travers le développement d'un projet de jeu complexe, le Reversi/Othello. Ce jeu, ancré dans la tradition des jeux de stratégie, est connu sous le nom d'Othello dans sa version commerciale et se caractérise par une lutte intense entre deux adversaires, symbolisés par les couleurs noir et blanc. L'objectif est de créer un programme informatique capable de simuler ce jeu, en mettant en œuvre des concepts avancés de programmation, d'intelligence artificielle, et de gestion d'interface utilisateur.

# 2. Fonctionnement du programme

La structure du programme est articulée autour de plusieurs classes fondamentales, organisées en deux groupes principaux. Le premier groupe prend en charge la gestion globale du jeu, tandis que le second groupe est dédié à la gestion des joueurs. Cette approche permet une interaction harmonieuse entre les différentes composantes du programme, facilitant ainsi la simulation cohérente d'une partie de Reversi.

## 2.1 Gestion du jeu

**Board :** La classe est conçue pour gérer et représenter le plateau de jeu, ainsi que pour effectuer diverses opérations telles que l'initialisation, l'évaluation des scores (**evaluate_score**), la recherche des mouvements légaux possibles(**get_legal_moves**), l'ajout de pions(**add_pawn**), l'affichage du plateau(**display_board**), et la détermination

de la fin de la partie(**isGameOver**). Le constructeur **Board** initialise le plateau avec les positions initiales des pions pour les deux joueurs, en utilisant une représentation sous forme de dictionnaire (map) pour stocker les informations sur les cases et les pions. Les pions du joueur 1 ('X') et du joueur 2 ('O') sont disposés au départ, et les cases vides sont représentées par le caractère '.'. Le nombre initial de pions est également défini à 4.

**Game :** La classe gère l'initialisation des joueurs, le démarrage de la partie, et la conclusion de celle-ci. La méthode **initializePlayers** permet à l'utilisateur de définir le type de joueurs (humain, IA, ou à partir d'un fichier) pour les joueurs X et O, en demandant les informations nécessaires telles que le nom ou le chemin du fichier. La méthode **startGame** démarre la partie, sollicitant les mouvements des joueurs à tour de rôle jusqu'à la fin du jeu. Elle utilise des instances des classes dérivées de la classe de base **Player**. Enfin, la méthode **endGame** affiche les scores finaux et libère la mémoire allouée pour les objets joueurs, concluant ainsi la partie. Le code est également accompagné d'inclusions de bibliothèques pour les opérations de gestion des fichiers, des temporisations, et d'autres fonctionnalités nécessaires à la simulation d'une partie de Reversi.

## 2.2 Gestion des joueurs

**Player :** La classe **Player** forme la base du système de joueurs dans le projet de jeu Reversi. Son constructeur initialise le joueur avec son nom, la couleur de ses pions, et établit son état initial en indiquant s'il a passé son tour. Les méthodes de traduction permettent de convertir les coordonnées des mouvements entre les représentations entières et en chaînes de caractères lisibles par l'utilisateur, grâce à une table de correspondance. La méthode **legal_moves** génère et affiche les mouvements légaux possibles, les traduisant en coordonnées compréhensibles, et renvoie également une

liste de ces mouvements en format de chaînes. Enfin, les méthodes **get_pass** et **get_name** sont des utilitaires essentiels, la première renvoyant l'état du joueur concernant le passage de son tour, et la seconde retournant simplement le nom du joueur.

**Humain :** La classe **Humain** constitue une composante essentielle du projet de jeu Reversi, héritant de la classe de base **Player**. Son constructeur initialise les attributs spécifiques au joueur humain, tels que le nom, la couleur des pions, et le type de joueur. La méthode **verify_answer** garantit la validité des réponses saisies par le joueur, en vérifiant la conformité aux règles du jeu et en comparant les réponses aux mouvements légaux possibles. Enfin, la méthode **play** gère le déroulement du tour de jeu pour un joueur humain, affichant le plateau, présentant les mouvements légaux, et sollicitant l'entrée du joueur. Elle prend également en compte le cas où le joueur choisit de passer son tour, gérant les affichages correspondants.

**Ai :** Cette classe représente un joueur virtuel qui prend des décisions automatisées pour effectuer des mouvements pendant le jeu. Le constructeur initialise les attributs de la classe, tels que le nom, la couleur des pions, le niveau de difficulté, le type de joueur (IA), et détermine l'adversaire du joueur IA. La méthode **arbre** implémente l'algorithme minimax avec élagage alpha-bêta pour évaluer les différentes possibilités de mouvements, explorant un arbre de jeu jusqu'à une certaine profondeur voulue grâce au niveau indiqué préalablement. La méthode **play** guide le joueur IA dans le choix du meilleur mouvement en explorant l'arbre des possibilités, ajustant sa stratégie en fonction du niveau de jeu, et utilisant la classe **Analyser** pour évaluer les configurations du plateau. Le code inclut également des mécanismes pour adapter la stratégie de l'IA en fonction de la phase du jeu (début, milieu, fin). Les résultats de chaque analyse sont affichés en temps réel pendant le calcul, fournissant une vue du processus de prise de décision de l'IA.

**Analyser :** La classe **Analyser** est conçue pour évaluer un plateau de jeu à un instant donné et attribuer un score en fonction de différents critères tels que le nombre de pions, la disposition spatiale des pions, et les mouvements légaux disponibles. Le constructeur initialise les coefficients de pondération pour ces critères en fonction de trois phases de jeu distinctes : début, milieu et fin de partie. La méthode **Analyse** parcourt le plateau, comptabilise le nombre de pions pour le joueur actuel et son adversaire, évalue la disposition des pions en fonction d'un tableau de scores prédéfini, et compte les mouvements légaux disponibles. Le score final est calculé en combinant ces différentes composantes avec les coefficients de pondération appropriés. Les méthodes supplémentaires **set_sco_EARLYGAME**, **set_sco_MIDDLEGAME**, et **set_sco_ENDGAME** permettent de modifier dynamiquement les coefficients en fonction de la phase du jeu.

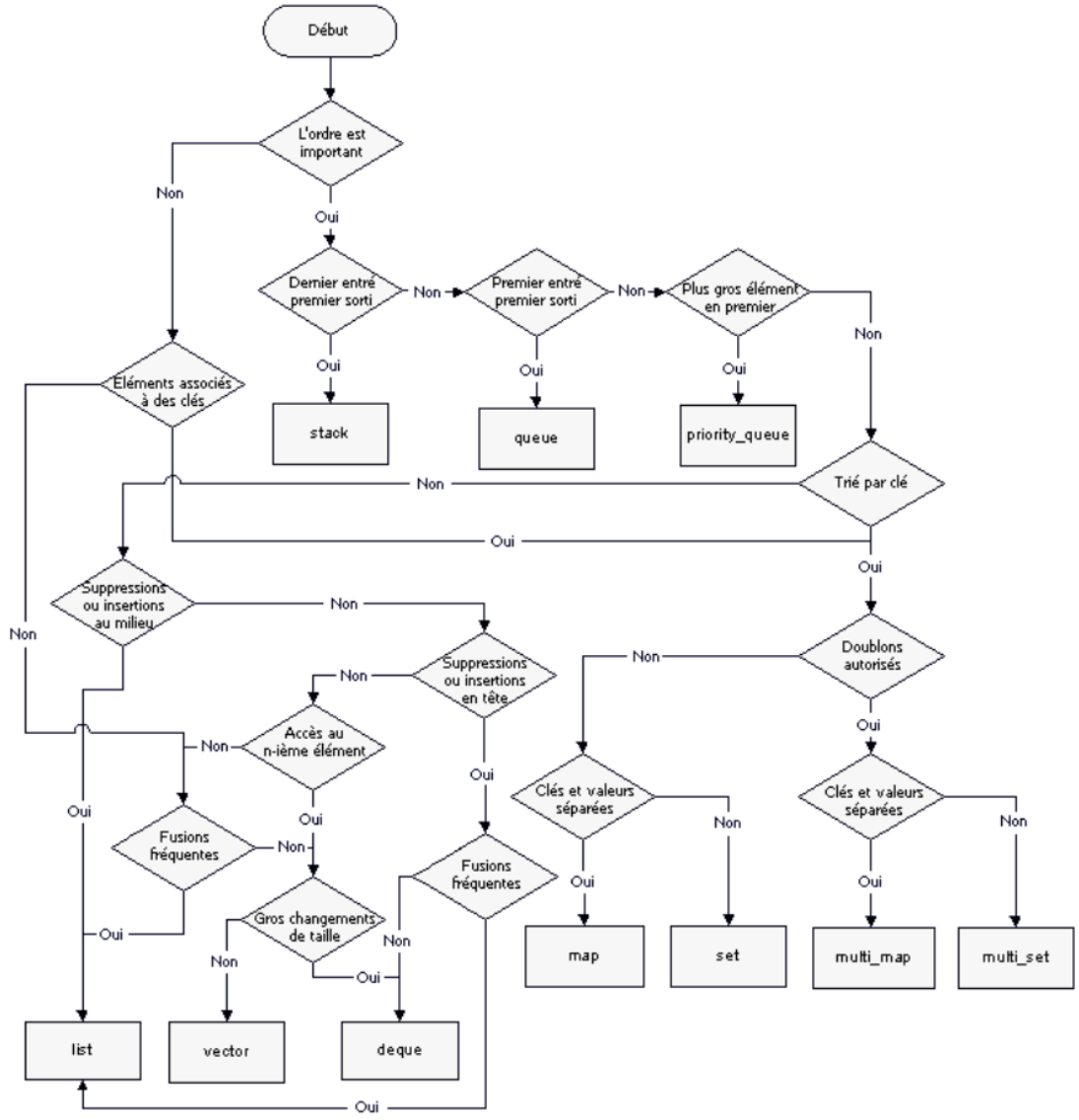
**Fichier :** Cette classe représente un joueur qui lit et écrit des mouvements dans des fichiers texte pour communiquer avec un adversaire. Le constructeur initialise le nom du fichier à lire et à écrire en fonction de la couleur du pion attribué au joueur. La méthode **initNom** tente d'ouvrir les fichiers et d'attendre que l'adversaire écrive son nom dans le fichier correspondant. La méthode **advMove** envoie le mouvement de l'adversaire dans le fichier, et la méthode **verify_answer** s'assure de la validité de la réponse reçue. La méthode **play** attend le mouvement de l'adversaire dans le fichier, vérifie sa validité, et renvoie le mouvement traduit en coordonnées du plateau. La classe utilise des fichiers texte pour synchroniser les mouvements entre les joueurs et attend que l'adversaire écrive avant de lire. Cette implémentation permet la communication entre deux instances du jeu Reversi via des fichiers texte partagés.

# 3. Choix de la structure de donnée

Dans la représentation du plateau de jeu 8x8, une structure de données a été utilisée pour modéliser chaque case du plateau de manière efficiente. Une map associant chaque case à une lettre 'X' ou 'O' a été employée, où les lettres représentent les pions des joueurs, et un nombre allant de 11 à 88 est utilisé pour identifier chaque case. Cette numérotation spécifique est organisée de manière que les dizaines représentent les lignes du plateau, allant de 1 à 8, et les unités représentent les colonnes, également de 1 à 8. L'équipe a étudié la complexité des différentes structures de données disponibles dans la librairie standard, concluant que la map était le meilleur choix pour représenter le plateau de jeu, offrant des avantages tels que la correspondance clé-valeur, le tri automatique, une recherche efficace, et l'absence de doublons autorisés, alignant ainsi la structure avec les besoins spécifiques du jeu Othello.


*Représentation du plateau*

La sélection d'une map pour modéliser le plateau 8x8 du jeu a été guidée par des considérations clés. La map offre un ordre automatique basé sur les clés, crucial pour les opérations de recherche. De plus, l'assurance de l'unicité des clés évite les doublons indésirables. La séparation nette entre clés (positions sur le plateau) et valeurs (pions 'X' ou 'O') simplifie la manipulation des données. (Voir choix du conteneur)




*Choix du conteneur (document fourni par l'assistant)*

Les choix cruciaux de la structure de données utilisée ont été effectués dans la conception de la classe IA, ils ont été utilisés pour représenter conceptuellement l'arbre de jeu. Une idée initiale était de créer un arbre de données sur le tas en utilisant l'allocation dynamique de mémoire. Cependant, cette approche s'est révélée moins performante en raison de la gourmandise associée à la gestion de la mémoire dynamique. L'allocation et la libération continues de mémoire ont entraîné des ralentissements significatifs, impactant la rapidité globale de l'algorithme.

Parallèlement, une autre approche a été explorée, consistant à modéliser l'arbre de jeu à l'aide d'une fonction récursive. Contrairement à l'allocation dynamique sur le tas, cette méthode tire parti de la gestion automatique de la mémoire dans la pile. Les nœuds de l'arbre sont ainsi empilés et désempilés au fur et à mesure de l'exécution, évitant ainsi les coûts associés à l'allocation dynamique. Cette approche s'est avérée significativement plus rapide, démontrant l'importance de choisir une structure de données adaptée aux besoins spécifiques du problème et aux contraintes de performance.

# 4. Choix de l'algorithme

Le choix de l'algorithme pour la prise de décision de l'IA a été dicté par la nécessité d'optimiser le temps de jeu tout en maintenant une stratégie efficace. Nous avons opté pour l'algorithme Mini-Max avec l'élagage Alpha-Beta en raison de sa capacité à explorer efficacement l'arbre de recherche des mouvements possibles tout en réduisant la complexité temporelle. La contrainte de temps de jeu de l'IA, fixée à 20 secondes, a conduit à une hauteur d'arbre limitée à 6 niveaux. Cette hauteur a été soigneusement choisie pour garantir que l'IA prenne des décisions réfléchies dans le délai imparti, tout en maintenant un équilibre entre la qualité de la prise de décision et les ressources computationnelles disponibles. L'utilisation de l'élagage Alpha-Beta dans l'algorithme Mini-Max offre une amélioration significative des performances en évitant

de calculer tous les mouvements possibles. Cet élagage permet d'anticiper les branches de l'arbre de recherche qui ne sont pas prometteuses, réduisant ainsi considérablement le nombre de nœuds évalués. Concrètement, l'algorithme ne poursuit pas l'exploration des branches qui ne peuvent pas influencer la décision finale, accélérant ainsi le processus de prise de décision de l'IA. Cela se révèle particulièrement avantageux dans des jeux tels que le nôtre, où le nombre de mouvements possibles peut être élevé.

# 5. Conclusion

En conclusion, le jeu fonctionne parfaitement lors de nos essais personnels, démontrant l'efficacité des choix algorithmiques et structuraux faits. Cependant, il est important de souligner les défis rencontrés lors des tests fournis par le professeur. Ces difficultés ont conduit à des ajustements nécessaires de l’implémentation. En fin de compte, cette expérience a renforcé notre compréhension du domaine et notre capacité à relever des défis algorithmiques, soulignant l'importance continue de l'itération et de l'amélioration constante dans le développement logiciel.