ULB UNIVERSITÉ LIBRE DE BRUXELLES

ECOLE POLYTECHNIQUE DE BRUXELLES

INFO-H304: Compléments de programmation et d'algorithmique

Projet Reversi

Arthur Elskens
Ammar Hasan
Manal Ouahhabi

Professeur Jérémie Roland

18 Décembre

Année académique 2020-2021

Table des matières

1 Introduction

Dans le cadre du cours de compléments de programmation et d'algorithmique, les étudiants sont amenés à réaliser un jeu de type Reversi à l'aide du langage C++. Pour ce faire, il a fallu adopter une architecture adéquate, sélectionner la structure de données correspondant le mieux aux besoins du jeu ainsi que les algorithmes les plus efficaces. Les divers choix qui auront permis l'aboutissement de ce projet seront discutés dans ce rapport.

2 Architecture du programme

Comme tout programme orienté objet, ce projet Reversi est scindé en différentes classes, comme l'illustre le diagramme de classes (cf Annexes) ...

La classe Main permet le lancement du jeu, et d'assurer la continuité des tours jusqu'à ce que la partie se termine. La classe Map est chargée, quant à elle, de la bonne application des règles du jeu et de déterminer le gagnant de la partie. View est uniquement consacrée à l'affichage du plateau dans le terminal. La classe Player permet la participation de différents types de joueurs, humain, intelligence artificielle ou encore l'interaction avec le joueur via un fichier indépendant. Une classe propre aux pions n'a pas été jugée nécessaire. Ici, la position de chaque nouveau pion est enregistrée en tant que chaîne de caractères et est associée au joueur actif par sa valeur entière 79 pour le joueur blanc et 88 pour le joueur noir.

3 Structure de données

3.1 Map

Le type de données abstraites désiré pour le conteneur du plateau de jeu est: insert, search et delete. Dans cette veine, l'utilisation d'un arbre binaire de recherche est des plus intéressante par son temps de calcul en $O(\log n)$ pour l'ensemble du type de données abstraites voulu. Dans la librairie standard du C++, il existe plusieurs implémentations des arbres binaires de recherche, celle qui a été choisie est le conteneur **map**. Celui-ci fonctionne à l'image d'un dictionnaire de Python avec un système de clé et de valeur associée, ce qui en fait d'ailleurs un conteneur trié. Dès lors, il est utilisé afin de pouvoir enregistrer les positions entrées par un joueur. En effet, grâce à sa particularité de stocker des paires "clé-valeur", la clé sera la position sous forme de *string* et la valeur sera la couleur du joueur, sous forme de *int*.

3.2 Set

Lorsqu'un joueur écrit "help" dans le terminal ou dans le fichier texte correspondant aux mouvements qu'il veut jouer, une série de positions s'affichent. Ces mouvements sont les uniques positions que le joueur actuel peut effectuer. Le conteneur qui a été utilisé pour stocker ces information est un **set**, ce conteneur est implémenté dans la librairie standard du C++ à l'aide d'arbres binaires de recherche. La raison de ce choix est basée sur deux propriétés du set:

- Valeurs triées: Ceci est important pour que le joueur ayant entré la commande "help" puisse voir de façon structurée tous ses mouvements possibles.
- Unicité des valeurs: L'algorithme permettant de trouver les positions acceptables n'est pas sélectif quant aux éventuels doublons de positions renvoyés. L'utilisation d'un set permet dès lors de n'avoir qu'une seule fois la position stockée malgré plusieurs insertions similaires.

3.3 Vector et unordered_set

Le temps de calcul de la meilleure position à jouer déterminée par l'algorithme d'élagage alpha-bêta utilisé pour l'implémentation de l'AI (cf. section 4.3) est particulièrement sensible

à l'ordre dans lequel les mouvements possibles sont analysés. Pour donner un meilleur temps moyen de calcul, analyser les mouvements dans un ordre aléatoire est efficace.
[Russell and Norvig, 2009]

Dans ce but, de façon similaire au cas du set, un **vector** est utilisé dans un premier temps pour stocker les positions possibles d'un joueur, ceci de façon non ordonnée et sans se soucier des éventuels doublons de positions. Ensuite, le vector est mélangé aléatoirement à l'aide de la fonctionnalité "shuffle()" de la librairie standard du C++. Finalement, le vector est parcourut de part en part pour implémenter chaque valeurs dans un **unordered_set**. Le unordered_set est comme son nom l'indique un set dont les éléments ne sont pas triés, il sera utilisé de la même façon que le set mais uniquement pour l'algorithme de l'AI.

4 Types de joueur

Comme dit précédemment, le jeu fait intervenir trois types de joueurs différents, qui seront étudiés dans cette section.

4.1 Humain

Le joueur de type humain lit simplement les entrées dans le terminal, et si elles sont viables, sont jouées. Cette viabilité est étudiée dans la classe Map. Le terminal demande d'entrer une nouvelle position tant que celle-ci n'est pas correcte.

4.2 Fichier

Le joueur de type fichier permet au jeu de communiquer avec un fichier. Lorsqu'un joueur folder est instancié, le jeu ouvre deux fichiers, le premier en lecture et le second en écriture. Dans le fichier ouvert en lecture le jeu invite le joueur à entrer la position voulu. Dans le cas d'une position non valide entrée, le jeu demandera sur la sorti standard d'entrer une nouvelle position dans le fichier et ce jusqu'à ce que la position soit valide. Dans le fichier ouvert en écriture, le jeu écrira les coups joués par l'opposant du joueur folder. Pour pouvoir communiquer avec d'autre jeu, le programme attendra qu'un fichier de lecture soit crée, il ne le créera pas par lui même. Les noms des fichiers sont "blanc.txt" et "noir.txt". Ils seront ouvert soit en lecture soit en écriture selon la couleur du joueur folder.

4.3 Intelligence Artificielle

Dans un jeu tel qu'Othello, l'implémentation d'une intelligence artificielle peut se faire de différentes façons: avec un algorithme "branch and bound" ou encore à l'aide de machine learning. La méthode "branch and bound" a été ici explorée.

4.3.1 Minimax

La recherche par Minimax est une prémière méthode de branch and bound possible pour réaliser l'AI, celle-ci consiste en la réalisation d'un arbre de décision dans lequel tous les mouvements possibles vont être étudiés. Cet algorithme permet de parcourir tour à tour jusqu'à un état final les choix du joueur Max et du joueur Min, le joueur Max étant ici l'AI et le joueur Min son adversaire. Le but de Max, comme son nom l'indique, est de trouver l'état de jeu qui va (selon certains standards) maximiser une utilité dans ce cas-ci un score, celui de Min est bien évidemment de le minimiser. [Russell and Norvig, 2009]

4.3.2 Alpha-beta pruning

Cette méthode Minimax intéressante pour trouver la meilleure position à jouer est cependant naïve dû au nombre exponentiel de chemins à explorer ($O(b^m)$). Une version plus efficace connue sous le nom d'élagage alpha-bêta existe, celle-ci n'est autre qu'une amélioration de la recherche par Minimax. Grâce aux paramètres alpha et bêta qui sont les meilleures valeurs trouvées jusqu'à présent respectivement dans les chemins Max et Min. Les branches qui par comparaison à ces derniers sont élaguées, sont jugées comme inaptes à fournir un meilleur score que les branches déjà explorées. Ainsi, l'exploration de l'arbre de recherche en est diminué. Ayant des avantages de temps de calculs, cette méthode a été implémentée dans ce projet. [Russell and Norvig, 2009]

Pour tirer un profit maximal de l'algorithme, quelques notions importantes ont été investiguées:

- **L'ordre de recherche:** Cet algorithme étant très sensible à l'ordre de recherche permet dans le meilleur cas de diminuer l'arbre de moitié, le nombres de chemins à explorer dans ce cas étant en $O(b^{m/2})$. Comme mentionné dans le section 3.3, changer aléatoirement l'ordre des mouvements à analyser permet d'améliorer le temps général de calcul de l'algorithme donnant le nombre total de chemins à explorer en $O(b^{m/4})$. Ce mélange aléatoire possède un avantage supplémentaire qui est de pouvoir jouer des parties toujours différentes contre l'AI.
- **La profondeur:** Pour que l'algorithme puisse arriver à une conclusion dans un temps

raisonnable, il n'est pas réaliste de considérer un état final comme une fin de partie. C'est pourquoi l'ajout d'un paramètre supplémentaire correspondant à la profondeur jusqu'à laquelle l'algorithme doit analyser les possibilités a été implémenté. Au plus cette profondeur est grande au plus l'algorithme peut prendre en compte différents chemins, mais également au plus le temps de calculs pourrait être long. Au vu des spécifications dans l'énoncé du projet, l'AI doit jouer un mouvements endéans les 20 secondes. Ainsi, après plusieurs essais, une valeur de 5 pour la profondeur a été adoptée, celle-ci maximise le nombre de tours à analyser tout en respectant le temps imparti imposé.

- **Les fonctions d'heuristiques:** Comme cela a été mentionné plus haut, une fois arrivé à un état final une utilité est renvoyée sous forme de score. Pour améliorer la prise de décision de l'algorithme, c'est sur ce calcul du score qu'il faut se pencher. Le calcul de l'heuristique peut se faire de différentes façons: par une méthode statique ou par une méthode dynamique. L'approche statique se base sur une matrice pour laquelle chaque position à un poids bien défini. Ce poids reflète l'importance de capturer une position, par exemple les coins ont un poids très grand comme ils ne peuvent pas être capturés par l'adversaire. Cette méthode efficace ne prend cependant pas en compte les changements du plateau de jeu ce qui la limite grandement. Une seconde approche, la dynamique, se base sur plusieurs principes dont la parité des jetons (différence entre les jetons de Max et de Min), la capture des coins (importance donnée à la capture des coins), la stabilité (une position est stable si on ne peut pas la capturer, semi-stable si elle ne peut pas être prise dans le prochain tour et instable si elle peut être prise dans le prochain tour) et la mobilité (la différence entre le nombre de mouvements que possède chaque joueur) tous décrit plus en détails dans [Sannidhanam and Annamalai, 2015]. Pour ce projet, après de nombreux essais, un mélange entre les deux méthode a été implémenté. L'heuristique d'un état se calcule donc par une combinaison linéaire de la parité des jetons, de la mobilité actuelle des joueurs, de la capture de coins et de la pondération de positions du plateau de jeu.

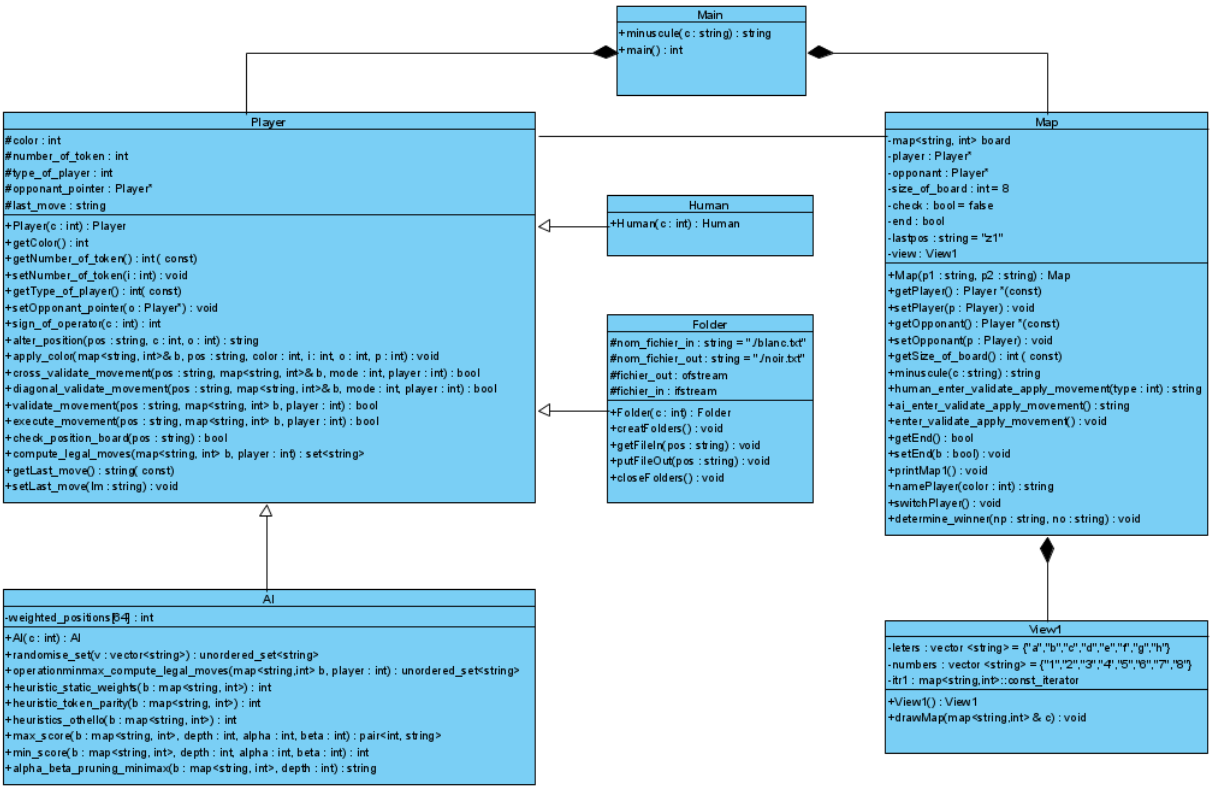
5 Conclusion

L'aboutissement de ce projet a permis de mettre en application divers aspects de la programmation : le choix d'une structure de données adéquate, ainsi que la mise en place d'une intelligence artificielle. Ce jeu peut certainement être amélioré avec la réalisation d'une interface graphique, afin de le rendre plus agréable visuellement. De plus, une étude plus approfondie des heuristiques d'Othello permettrait un meilleur choix de la part de l'AI. En effet, l'implémentation de la stabilité dans le calcul de l'heuristique dynamiserait tout à fait son calcul

et donnerait de meilleur résultats.

6 Annexes

6.1 Diagramme de classes



Liste de références

[Russell and Norvig, 2009] Russell, S. J. and Norvig, P. (2009). *Artificial Intelligence: A Modern Approach*. Prentice Hall, Third Edition. p.161-175.

[Sannidhanam and Annamalai, 2015] Sannidhanam, V. and Annamalai, M. (2015). *An Analysis of Heuristics in Othello*. Washington University, Department of Computer Science and Engineering.