Procédure de remise

Wenhao LUO

November 23, 2022

Contents

1 Introduction

En suivant le cours **8INF878-Intelligence Artificielle** de l'Université de Québec à Chicoutimi, on a réalisé ce projet "compétition d'agent intelligents pour le jeu d'échecs" par l'implémentation d'une intelligence artificielle comme un engine jouable sur la plateforme ARENA.

Ce document sert à introduire le fonctionnement de ce programme, à expliquer comment on a réalisé cette implémentation et à discuter les difficultés rencontrés pendant la réalisation de ce projet.

2 Fonctionnement du programme

2.1 Informations générales

Le programme réalisé, nommé `OCAP` est un engine qui est capable de jouer le jeu d'échecs avec l'aide du logiciel ARENA. Il est capable de comprendre une partie des commandes définies avec le protocole UCI, qui le permet de reconstituer, d'analyser et de jouer un jeu d'échecs avec les données transmis par `stdin` et `stdout`.

2.2 Commandes acceptées

Voici la liste de toutes les commandes qui peuvent être insérées et comprises par ce logiciel :

- `uci` : en acceptant cette commande, le programme retourne son identifiant et son auteur ;
- `isready` : en acceptant cette commande, le programme retourne le message "isready" qui permet de confirmer qu'il est prêt au calcul ;
- `go` : en acceptant cette commande, le programme commence à effectuer les calculs ; il devra retourner un résultat plus tard avec un délai prédéfini ;
- `stop` : en acceptant cette commande, le programme arrête son calcul et il retourne un résultat le plus vite possible ;
- `quit` : en acceptant cette commande, le programme arrête son fonctionnement et termine l'exécution de tous les threads ;
- `position startpos [moves] <move1>, <move2>, ...` : en acceptant cette commande, le programme reconstitue un damier au sein du programme, qui permet de réaliser les calculs plus tard.

2.3 Interaction avec ce programme

Avec l'interface ARENA, une personne peut jouer un jeu d'échecs avec ce programme en le chargeant comme un des engines de l'ARENA. C'est également possible de jouer le jeu directement en lançant ce programme dans une ligne de commande par insérer les commandes listées dans la section 2.2.

3 Implémentation

3.1 Langage et paradigme utilisé

Le programme est écrit en Java, et naturellement avec le paradigme OOP. Certaines parties sont écrites avec multithreading.

3.2 Topologie du programme

Le programme peut être séparé en trois parties différentes :

- un damier et des pièces de l'échecs construct au sein du système, qui permet de reconstruire un scénario du jeu et l'évaluer ;
- un programme qui permet d'analyser le meilleur choix dans une situation donnée, et de prendre des décisions ;
- un parseur minimise qui permet de comprendre les commandées entrées et de communiquer avec les autres composantes du programme.

Prochainement on va parler des trois parties à l'ordre.

3.3 Damier et pièces

La première partie du programme réalise un damier de l'échecs `Board` avec des pièces qui peuvent être mises sur un damier. Un damier contient une liste de pièces et leur coordonnées, qui est un "état" du jeu. J'ai également implémenté les fonctions qui permettent de réaliser les fonctionnements basiques, comme une fonction qui retrouve une liste des actions légales, et une autre fonction qui permet de réaliser une transition entre deux états. Les exemples peuvent être retrouvés dans le listing 1.

Des pièces du jeu sont programmés avec un héritage de classe : toutes les pièces sont considérées comme un membre de la classe `Piece`, ensuite chaque pièce réalise ses propres méthodes. Le code de la classe `Piece` est disponible comme le listing 2.

Le problème de gestion de mémoire a été pris en compte pendant le développement. Notons que dans la classe `Board` les pièces ne sauvegardent pas leur position sur le damier ; en fait il ne sauvegarde non plus l'objet `Board`. J'ai choisi cette implémentation afin d'éviter de recopier les objets `Piece` entre la transition des deux damiers.

Si un nouveau état du jeu doit être créé en déplaçant une pièce sur un damier, alors seulement la position des pièces sera modifié ; les différents damiers partagent les mêmes pièces : chaque fois on trace la position des pièces par un nouveau dictionnaire et tous les dictionnaires pointent les mêmes objets. Une illustration est donnée comme le figure 1.

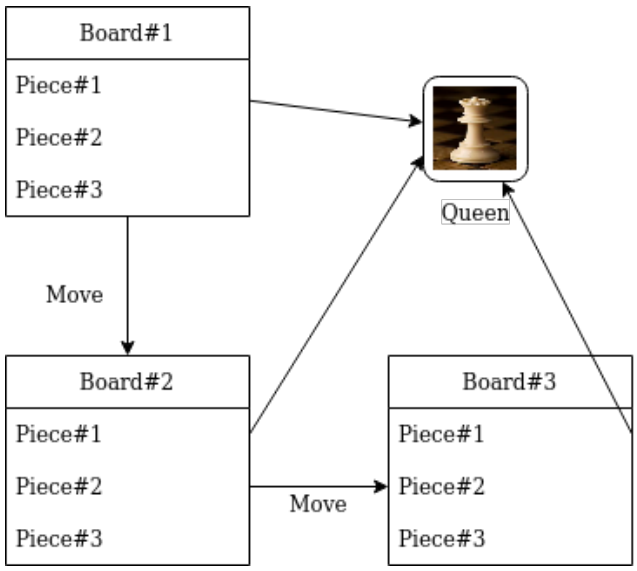

Figure 1: Illustration sur la création des nouveaux états.

La classe `Board` génère toutes les possibles actions à partir des positions des pièces. Les actions sont nommées comme `Move`. Un damier peut généré un nouveau damier selon une instance de `Move` donnée. On est donc capable de décrire les différents états et leur transition.

3.4 Analyseur

3.4.1 Algorithme utilisé

On utilise une recherche en appliquant un algorithme élagage alpha-bêta[1]. Le principe de cet algorithme est d'explorer les différents nœuds par profondeur, en applicant une optimisation supplémentaire en mémorisant deux valeurs α et β. Ces valeurs représentent les meilleurs/pires cas possibles. Si le résultat retrouvé par un nœud dépasse ces valeurs, alors théoriquement ces nœuds seront jamais visités, supposant que les joueurs ne choisissent que les meilleurs choix. Ainsi on évite de faire les visites non nécessaires.

Type	Valeur
Pion	1
Cavalier	3
Fou	3
Tour	5
Dame	9

Table 1: Les notes en fonction des pièces.

3.4.2 Fonction d'évaluation

Une fonction d'évaluation est nécessaire pour déterminer si un état est "bon" ou "mauvais". C'est le facteur plus important sur la qualité de notre programme. Heureusement il y a déjà les recherches sur ce domaine, et on a un système d'évaluation classique qui affecte une note sur chaque pièce sur un damier. C'est le *chess piece relative value*[2]. Selon ce système, on affecte une note de la manière présentée dans le tableau 1. D'ailleurs, puisque la perte des "Roi" cause la perte du jeu, on lui affecte une note de ∞. Les notes sont présentés en nombre flottant ; la valeur d'infinie est donc bien définie dans le contexte.

C'était la première implémentation que on a réalisée. Pourtant on trouve que cette implémentation ne permet pas de distinguer la qualité des différents choix au début du jeu : théoriquement l'échange des pièces n'est pas souvent spécialement au début du jeu, et cette évaluation ne peut pas trouver le meilleur choix dans ce cas. Pour résoudre ce problème, on a introduit une nouvelle fonction heuristique. Soit d_1 la distance entre une case et l'arête `a1-h1` et d_2 la distance entre une case et l'arête `a8-h8`, h_c un constant. La fonction h est définie par l'équation 1.

$$h = \begin{cases} h_c \, |d_1 - 5| & si \ p \in White \\ h_c \, |d_2 - 2| & si \ p \in Black \end{cases} \tag{1}$$

Autrement dit, la plus avancée une pièce est, la meilleure note qu'elle est affectée. Ceci permet de progresser le jeu et d'éviter les boucles. Après les tests, on a décidé de mettre $h_c = 0.1$.

3.4.3 Application multithread

L'algorithme alpha-bêta est implémenté sous la forme d'une fonction récursive. La fonction de exploration fait son calcul de façon continue. On a besoin de lancer cette fonction par un thread séparé, afin d'arrêter le calcul quand l'interface émet le signal de terminaison. C'est-à-dire le commande `stop` discutée dans la section 2.2.

On réalise la synchronisation par une variable partagée entre plusieurs threads. Une variable booléenne `isFinish` est déclarée au moment de l'instanciation. Le thread principal peut modifier sa valeur quand la recherche est terminée

; le thread qui réalise la recherche vérifie la valeur de cette variable quand il fait l'appel récursive. Ce mécanisme permet donc de terminer l'exploration instantanément.

Si on interrompe le thread au milieu d'une exploration, alors l'exploration n'est pas complète. Le résultat n'est pas fidèle. Pour résoudre ce problème, l'exploration est faite de façon progressive. On commence par un profondeur de k, une fois le calcul est fini on sauvegarde le résultat et on continue par un profondeur de $k+1, k+2, \cdots$. Le code est présenté dans le listing 3.

3.5 Communication

4 Difficultés rencontrés

5 Conclusion

References

[1] Wikipedia contributors. Alpha–beta pruning — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Alpha%E2%80%93beta_pruning&oldid=1115399006, 2022. [Online; accessed 23-November-2022].

[2] Wikipedia contributors. Chess piece relative value — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Chess_piece_relative_value&oldid=1122712884, 2022. [Online; accessed 23-November-2022].

Appendices

A Code implémenté

```
public class Board implements Cloneable {

    private final double heuristicConst = 0.1;

    private HashMap<PieceType, ArrayList<Piece>> availablePieces =
    new HashMap<>();
    private TreeMap<Coord, Piece> board = new TreeMap<>();

    private boolean isBlackMove = false;

    public Coord enpassant = null; // null means no "enpassant"
    // otherwise is the last coord of the "enpassant"
    // these variables are used to track the "castling"
    // zero for black, one for white
    private boolean[] isKingMoved = {false, false};
    private boolean[] isLeftRockMoved = {false, false};
    private boolean[] isRightRockMoved = {false, false};

    public boolean isBlackMove() {...}
    public TreeMap<Coord, Piece> getWhitePieces() {...}
    ...
}
```

Listing 1: Une partie des fonctions réalisées dans la classe `Board`.

```
package Board;

import java.util.TreeSet;

import mUtil.Coord;

public abstract class Piece {
    protected boolean isBlackVal;
    abstract public TreeSet<Coord> getLegalMoves(Board currentBoard
    , Coord currentCoord);

    abstract public String getShortName();

    public Piece(boolean isBlack) {
        this.isBlackVal = isBlack;
    }

    public boolean isBlack() {
        return isBlackVal;
    }

}
```

Listing 2: La classe abstracte `Piece`.

```

```

```
@Override
public void run() {
    int depth = 1;
    while (!isFinish && depth <= depthLimit) {
        explore(depth);
        depth += 1;
        System.out.format("info depth %d finish\n", depth);
        this.oldRoot = this.root;
        this.root = new Node(board);
    }
}
```

Listing 3: La recherche récursive, réalisée par un thread séparant.