Rapport du projet IA échecs

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.

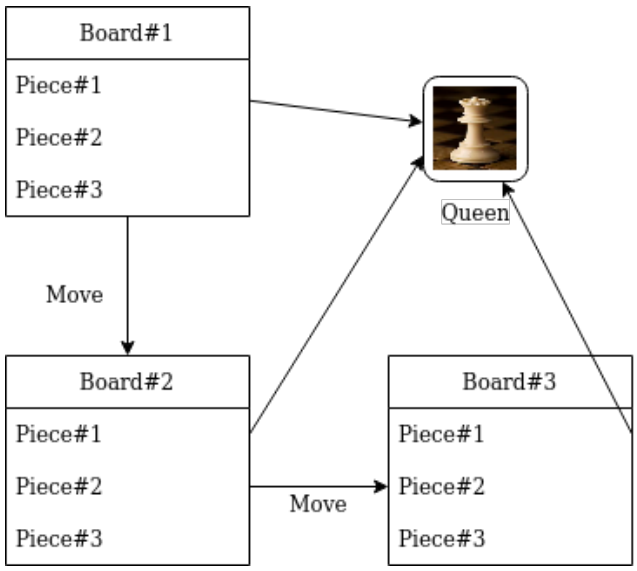

Figure 1: Illustration sur la création des nouveaux états.

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[3]. 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*[4]. 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. Le code est présenté dans le listing 3.

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 4.

3.5 Communication

3.5.1 Protocole UCI

Le protocole UCI, ou *Universal Chess Interface* est le protocole utilisé par ARENA qui permet de communiquer avec un engine d'échecs. L'ensemble de ce protocole est assez compliqué ; heureusement pour réaliser une interaction basique on n'a pas besoin de toutes les fonctionnalités. L'ensemble de commandes à traiter par ce programme est listé dans la section 2.2 ; le programme fonctionne.

Le format des commandes peut être considéré comme une liste de mots clés séparés par des espaces. Voici quelques exemples des commandes valides :

- `uci` ;
- `position startpos` ;
- `position startpos moves c2c4`.

3.5.2 Parseur minimist

Idéalement c'est meilleur de écrire un parseur qui analyse tous les tokens entrés de façon rigoureuse. Pourtant à cause de la limite du temps disponible on a choisi de créer un parseur très simplifié. Au lieu de parser toutes les entrées et de générer un AST, on traite les différentes entrées en fonction du premier mot clé reçu. En plus, puisque la plupart des commandes ne contient qu'un mot clé, le traitement ne concerne que ce premier mot clé dans la plupart du temps. Finalement on a un parseur réalisé sous la forme d'un branchement `switch`. Le code final est dans le listing 5.

3.5.3 Problème de la commande `go`

Parmi les commandes qui peuvent être analysées et répondues, la commande `go` introduit la problématique de la transition temporaire. En mettant cette commande comme l'entrée, le système doit effectuer un calcul puis donner un résultat avec un délai t. Spécifiquement, le délai défini dans le sujet est $t = 1s$. Une transition doit être faire au sein du système automatiquement s'il n'y plus

de entrée. La situation est plus compliquée en prenant compte d'une autre commande `stop`, qui peut forcer la terminaison du calcul.

La solution est l'introduction d'un nouveau thread et l'utilisation du mécanisme d'interruption. Le thread `main` lance le calcul sur un deuxième thread `analyse` ; il lance ensuite un troisième thread `notifier` qui détecte si une entrée est disponible de `stdin`. Le thread `main` exécute ensuite la commande `Thread.sleep(`t`)`, donc il peut donc :

- arrêter le calcul et retourner le bon résultat s'il n'y a pas d'interruption ;
- laisser le thread `notifier` lui interrompre et traiter le nouveau entrée en traitement le `InterruptedException`.

4 Difficultés rencontrés

4.1 Debugger avec une représentation graphique

Pendant le développement de la partie "Analyseur", on a pensé a implémenter une représentation graphique pour sauvegarder les nœuds explorés par le programme. Ceci facilite le processus de debugger. On a choisit de créer les fichiers `.dot` à partir du code Java pendant l'exécution. Ensuite les programmes de la famille Graphviz permet de créer une image `.PNG` ou `.SVG` ; un exemple généré par la ligne de commande `sfdp ./output.dot -x -Goverlap=scale -Tpng -o out.png` est lw figure 2. En pratique, cette solution n'est pas pratique avec un profondeur important. Notons que le figure 2 est la représentation avec le profondeur $p = 1$. Le nombre des nœuds augmente de façon exponentielle ; avec un profondeur de 3 le fichier `.PNG` généré peut avoir une taille de 6 Gb, et c'est difficile de suivre le raisonnement du programme en tout cas.

4.2 Nombreuses commandes disponibles avec UCI

Le premier problème que on a rencontré pendant le développement est le problème de la réalisation d'une interface UCI. Au début de développement on a pas assez de connaissance sur le fonctionnement d'ARENA, ainsi que comment ce logiciel peut interagir avec le code Java. Ensuite on a trouvé le fichier qui décrit le protocole UCI[2]. C'est un document daté de 2006 qui a proposé vingtaines commandes à traiter. Sûrement en réalisant toutes les commandes de façon robuste permet d'avoir un système de la meilleur fonctionnalité, mais on trouve après les essaies, alors que les commandes listées dans la section 2.2 sont nécessaires pour un système basique. En donc on a choisi implémenter ces commandes.

D'ailleurs, grâce à l'implémentation du syntaxe `switch`, ce possible d'ajouter plus de commandes dans le programme plus tard sans modifier la structure du programme.

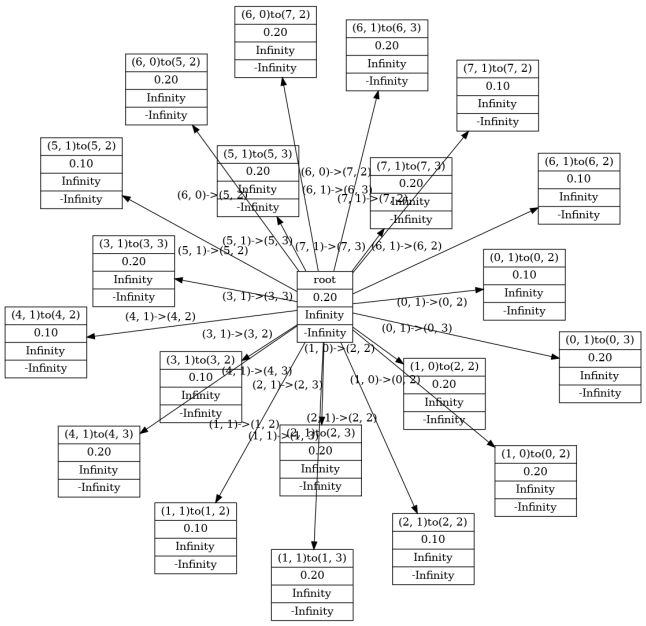

Figure 2: Une image généré par le programme Graphviz.

4.3 Problème avec l'exécution du code sur les plateformes Linux/Windows

Le développement de ce programme est fait totalement sur un système Linux Ubuntu 22 LTS. La différence entre un système Linux et un système Windows normalement ne pose pas de problème, ceci dit, il y a une différence non négligeable c'est le traitement d'un fichier donné. Le système Windows connaît les suffixes dans le nom d'un fichier, et il fait le bon logiciel exécuter les opérations. Donc sur un système Windows, le fichier `.jar` est un format valid pour un engine d'échecs pour l'ARENA. Car en mettant le fichier dans une commande d'exécution, le code exécuté réellement est `java -jar xxx.jar`. Alors que sur un système Linux, la ligne de commande ne permet d'identifier un fichier `.jar`, ARENA ne peut pas "exécuter" un fichier de tel format.

Pour résoudre ce problème, la première solution qu'on a conçu est de retrouver un programme qui permet de compiler un fichier `.jar` et sortir un fichier exécutable. Malheureusement cette solution est difficile à cause de la manque de solution facile. Nous avons donc choisi une solution alternative : on a créé un programme C qui lance le fichier `.jar` en appelant la fonction `execvp`[1]. Le programme C est disponible dans le listing 7.

Un problème parasite est résolu avec la version de Java utilisé pour l'exécution. Le programme a été écrit avec une version récente de Java 19. Ainsi comment le fichier `.jar` a été généré. Il semble que le fichier `.jar` généré demande donc

une version de Java VM assez récente pour son exécution ; il faut préciser dans la ligne de commande `-enable-preview` afin d'avoir le programme fonctionner. C'est la raison pourquoi que le programme a échoué son exécution pendant la petite compétition réalisée à l'UQAC le 22 novembre. Heureusement, après une réflexion, je pense que la configuration d'ARENA permet de l'ajout des paramètres dans la ligne de commande[1].

4.4 Optimisation avec le mouvement du "Roi"

La dernière changement que on a effectué sur le programme est une optimisation. Nous avons remarqué que la qualité de la solution retrouvée est influencée fortement par le profondeur de recherche. Pour gagner le temps à une exploration plus profonde, on a utilisé `VisualVM` pour obtenir le profil du programme. On a remarqué que le programme a passé le temps pour l'analyse du mouvement de la pièce "Roi", car dans la version précédente, on a considéré que cette pièce ne devrait pas être capable de déplacer vers une case qui peut être attaquée. La réalisation de cette méthode, demande finalement de calculer toutes les mouvements possibles des pièces de l'autre joueur[2]. C'est donc un calcul très lourd. Après une réflexion, on a décidé de abandonner cette fonctionnalité, car théoriquement un système sain ne choisit pas le mouvement de "se suicider". Ce choix nous économise beaucoup de temps de calcul, permet de calculer maintenant un profondeur de 5 au lieu de 2.

Le changement introduit un nouveau problème, car maintenant l'engine ne peut pas retrouver le situation du pat[5]. Mais on suppose que cette situation n'arrive pas souvent. Donc pour l'instant on le laisse dans notre programme.

5 Conclusion

Dans ce document, on a présenté le programme qu'on a réalisé pour ce projet d'intelligence artificielle, le fonctionnement de notre programme, leur implémentation, ainsi les problèmes, les solutions, et les réflexions qu'on a expérimenté pendant la réalisation de ce projet.

Le rendu de ce projet est donc ce document avec un engine d'échecs, nommé OCAP. Les commandes qu'il est capable à traiter ont été données dans la section 2.2, et un bug connu est discuté dans la section 4.4. Le code de source est disponible sur GitHub : `https://github.com/LuoQuestionmark/projet-ia-echecs`.

Personnellement je suis content que le projet est terminé sans trop de difficultés et je suis fié avec le programme que j'ai écrit.

[1] Personnellement je n'ai pas encore testé son fonctionnalité sur un plateforme Windows, mais dans le pire cas le code Java est toujours disponible et vous pouvez faire la compilation à partir du code disponible sur `https://github.com/LuoQuestionmark/projet-ia-echecs`

[2] Techniquement pas tous les mouvements de leur "Roi", sinon il y aura une récursion infinie

References

[1] execvp(3) - linux man page. https://linux.die.net/man/3/execvp. [Disponible sur un système Linux en tappant man execvp.].

[2] Description of the universal chess interface (uci). http://download.shredderchess.com/div/uci.zip, 2006. [Une version digitale peut être retrouvée sur le site https://gist.github.com/DOBRO/2592c6dad754ba67e6dcaec8c90165bf].

[3] 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].

[4] 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].

[5] Wikipédia. Pat (échecs) — wikipédia, l'encyclopédie libre, 2022. [En ligne; Page disponible le 30-mars-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`.

```java
synchronized public double explore(int depth, Node current) {
    // the actual recursive body of function "explore"
    if (this.isFinish || depth == 0) {
        return current.calScore(true);
    }

    double val;
    Board newBoard;
    Node child;
    synchronized (lock) {...}
}
```

Listing 3: La variable `isFinish` et son utilisation sur la synchronisation.

```java

@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 4: La recherche récursive, réalisée par un thread séparant.

```java
private static void inputParser(String inputString) {
    String[] tokens = inputString.split("[ ]+");
    switch (tokens[0]) {
        case "uci":
            printUCI();
            break;
        case "isready":
            printIsReady();
            break;
        case "go":
            startAnalyser();
            break;
        case "stop":
            Move m = stopAnalyser();
            if (m == null)
                break;
            printBestMove(m);
            break;
        case "quit":
            if (notifier != null) {
                notifier.finish();
            }
            if (analyser != null) {
                analyser.finish();
            }
            System.exit(0);
            break;
        case "position":
```

```
            dealPosition(tokens);
            break;
        default:
            break;
    }
}
```

Listing 5: Le parseur qui traite les entrées.

```
public static void main(String[] args) throws IOException {
    bufferedReader = new BufferedReader(new InputStreamReader(
    System.in));
    startReadInput();
    while (true) {
        try {
            Thread.sleep(timeout);
            if (analyser != null) {
                Move m = analyser.getBestMove();
                printBestMove(m);
            }
        } catch (InterruptedException e) {
            while (bufferedReader.ready()) {
                String line = bufferedReader.readLine();
                inputParser(line);
            }
        }
    }
}
```

Listing 6: Le corps de la partie qui traiter les entrées/sorties.

```
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>

#define PATH_MAX 2000

extern int errno;

// java --enable-preview -jar projet-ia-echecs.jar

int main(int argc, char** argv) {
    // chdir("./build");
    char cwd[PATH_MAX];
    char* args[] = {"java", "--enable-preview", "-jar", "./projet-
    ia-echecs.jar", NULL};
    execvp(args[0], args);
    perror("");
}
```

Listing 7: Ce programme en C permet de lancer le fichier .jar par sa version compilée sur un système Linux.