Rendu du projet d'intro à l'IA

Nicolas Devatine, Julien Guyot

20 avril 2018

1 Introduction

Contenu de l'archive L'archive contient :

1. Les sources java, dans le dossier src.
2. Le document présent, résumant le contenu des sources, ainsi que le déroulement général du projet.
3. un dossier apidocs, qui contient la javadoc du projet

Résumé L'objectif de ce projet a été de modéliser le jeu du quarto et de réaliser une intelligence artificielle capable d'y jouer par la mise en place des techniques et des algorithmes vus en cours.

Dans ce rapport nous allons détailler et justifier les choix d'implémentation que nous avons fait au niveau de l'implémentation du jeu, des algorithmes ou encore de l'heuristique utilisée.

2 Description des problèmes rencontrés au cours du projet

2.1 Implémentation du plateau de jeu

Dans un premier temps, nous avions prévu un modèle assez optimisé, qui etait refléchie pour prendre le moins de place possible, et pour que les opérations sur le plateau soient le moins coûteuses en CPU possibles — Ce modèle était décrit assez précisément dans le document du premier rendu —

Nous avions imaginé une modélisation de l'état du plateau qui occupait "en principe" (c'est à dire hors questionnements spécifiques à java) une douzaine d'octets (un long, deux shorts, et un byte).

Néanmoins, cette idée nécessitait d'avoir un contrôle assez fin sur les entiers, comme par exemple la gestion des entiers non signés, ou une gestion propre des opérateurs bit à bit.

Malheureusement, la gestion de java des entiers (et/ou des shorts et bytes) de n'a pas permis la réalisation qui était à la base prévu. Cela est principalement dû aux spécificités de java suivantes :

- Les entiers java sont forcément signés — en tous cas les entiers atomiques ("int" et pas "Integer").
- Les opérateurs bit à bit traitent uniquement les entiers (c'est à dire ni les bytes, ni les short). Par conséquent, l'opération (byte & byte) (ET logique)
  1. Convertit les bytes en int
  2. Applique l'opérateur
  3. Renvoie un int, qu'il faut re-caster si l'on cherche à calculer un byte

Si cela ne semble a priori pas plus problématique que cela, nous avons observé des effets qui étaient insoupçonnés : Par exemple, l'opération (*byte*) $F0_{16}$ >>> 4 (où $F0_{16}$ est un byte), rendra FF_{16}, alors qu'on aurait pu s'attendre à $0F_{16}$ [1]

Dans un premier temps, l'optimisation semblait intéressante — quasiment pas d'espace gâché, des modification du plateau rapides, et par conséquent de bonnes performances — la réalité fut autre :

Le signage des entiers, allié au "mono-type" des opérateurs logiques causaient des effets inattendus — De la même nature que l'exemple évoqué ci-dessus — . Les divers calculs qui auraient été nécessaires pour pallier à ces problèmes auraient été assez difficiles à implémenter, et auraient considérablement ralenti les calculs sur le plateau. Par conséquent, ce qui aurait dû faire la force

Ainsi, nous avons choisi une implémentation du plateau moins extrémiste, sans être trop différente de la première idée, afin de ne pas avoir à réécrire trop de code.

3 Modélisation du quarto

Nous décrirons ici l'implémentation des diverses classes composant le package jeux.quarto

3.1 Plateau de jeu

La classe contenant le plateau de jeu est la classe PlateauJeu.

Description de la classe Le plateau est constitué de quatre à six attributs : Le plateau en lui-même (une matrice de byte) qui donne l'emplacement des pièces sur le plateau, un attribut dénotant des pièces qui ont été jouées, un attribut (byte) indiquant le cas échéant quelle pièce devra être jouée ainsi qu'un dernier attribut indiquant l'état du tour (quel joueur doit jouer quel type de coup), ainsi que les deux joueurs.

[1] X_N désigne le nombre X écrit en base N. L'opérateur >>> désigne le décalage à droite logique

Modélisation des pièces Chaque pièce est modélisée sous la forme d'un byte, qui contient les quatre caractéristiques définissant la pièce. Ce byte est de la forme

$$0000 \ \ couleur\ hauteur\ sommet\ forme$$

Et tel que:

- couleur = 1 si la pièce est bleue, 0 si la pièce est rouge
- hauteur = 1 si la pièce est grande, 0 si elle est petite
- sommet = 1 si le sommet est plein, 0 sinon
- forme = 1 si le sommet est carré, 0 si le sommet est rond

Attribut plateau Le plateau en lui-même est une matrice 4 $\times$4 de bytes, où chaque case peut accueillir une pièce. Si une case donnée ne contient pas de pièce, sa valeur est de -1.

Attribut piece_a_jouer L'attribut de la pièce à jouer est la pièce qu'il faut jouer, dans le cas où le tour est un dépôt de pièce. Ce dernier est initialisé à -1 , afin que lors de la première "double-action"[2]

Attribut indices_pieces Cet attribut est un int, et indique quelles pièces ont été jouées, ou données. Si le n-ème bit de poids faible est à 1, cela indique que la pièce d'identifiant $n-1$ a été donnée, ou posée.

Attribut etat_du_tour Comme son nom l'indique, cet attribut indique l'état du tour. Son bit de poids faible est à 0 si un des joueurs doit donner une pièce à l'autre joueur, et à 1 si l'un des joueurs doit poser une pièce.
Le second bit de poids faible est à 0 si le joueur j0 doit accomplir une action, et à 1 si c'est j1 qui doit jouer.

Note sur les joueurs Dans la classe PlateauJeu (et plus généralement dans le package quarto), nous avons considéré que le premier joueur à jouer (j0) était le joueur noir.
Par conséquent, toutes les documentations/tous les commentaires mentionnant le joueur noir/le joueur blanc parlent respectivement du joueur j0, et du joueur j1.

[2] on appelle double-action un coup composé à la fois d'un dépôt et d'un don de pièce. Ainsi, une "action" sera uniquement soit un dépôt, soit un don

3.2 Coups

3.2.1 Classe CoupQuarto

Cette classe — qui implémente l'interface CoupJeu — possède deux attributs, et est utilisée pour représenter une action qui peut être un don comme un dépôt.

Son attribut is_don est un booléen indiquant le type d'action représenté : S'il est à *true*, alors le coup est un don de pièce, dans le cas contraire le coup est un dépôt.

Dans le cas où l'action est un don, alors l'attribut idCoup prend la valeur d'un identifiant de pièce, dont la forme est décrite ci-dessus.

Dans le cas contraire, l'action est donc un dépôt, et idCoup représente un identifiant d'une coordonnée, de la forme

$$0000 \quad \textit{colonne ligne}$$

Où la colonne est entre 0 et 3, est est codée sur 2 bits, idem pour l'indice de la ligne.

La classe contient deux constructeurs, un à partir d'un byte et d'un booléen, et l'autre à partir d'une chaîne de caractères, dont la forme est spécifiée dans la documentation.

3.2.2 Classe DoubleCoupQuarto

Lors des tests sur les algorithmes de jeux, nous avons remarqué que les algorithmes de jeu partaient systématiquement du principe que les deux joueurs jouaient l'un après l'autre.

Par conséquent, afin de ne pas avoir à modifier des algorithmes qui, du coup, ne serait plus généraux, nous avons implémenté une classe implémentant CoupJeu, et permettant de faire en sorte que les joueurs jouent successivement l'un après l'autre.

Ainsi, cette classe ne contient plus un seul attribut "identifiant", mais deux attributs, qui indiquent l'identifiant de la coordonnée, et l'identifiant de la pièce.

Il y a toujours deux constructeurs, le premier par deux byte, le second par une chaine de caractères de la forme "[coordonnée]-[identifiant piece]".

3.3 Autres classes

Contient une procédure main, qui déroule une partie

HeuristiqueQuarto Contient une heuristique pour le quarto

JoueurQuarto Implémentation de l'interface IJoueur. Contient un AlgoJeu, qui définit quelle heuristique est adoptée.

JoueurHumainQuarto Implémente l'interface AlgoJeu, pour permettre à un humain de jouer au quarto en tapant les coups joués

PartieQuarto

4 Heuristique

Pour notre heuristique nous avons d'abord pris en compte les cas extrêmes qui sont les cas où la position est finale ("il existe une configuration gagnante pour au moins une caractéristique, pour l'un des deux joueurs"). Dans ce cas, nous fixons une heuristique maximale ou minimale, dans les cas où la partie est respectivement gagnante ou perdante?

Il reste ensuite à traiter les autres configurations. La difficulté du quarto est qu'en plus du plateau en lui-même, il faut prendre en compte les pièces non encore jouées qui vont avoir une influence sur l'heuristique. De plus comme un coup joué est en réalité un double coup, il faut à la fois traiter le plateau avec le dépôt qu'on a fait et le choix de la pièce à donner. Au quarto il est plus facile d'adopter des stratégies spécifiques en fin de partie, notre idée est qu'un plateau avantageux est un plateau qui va forcer l'adversaire à donner une pièce d'une certaine caractéristique en ne lui donnant pas le choix. Pour cela il faut mettre en "famine" certaines caractéristiques dans le choix des pièces restantes tel que en plaçant une pièce possédant leur caractéristique opposée on gagne la partie. On va alors associer une valeur à cela pour calculer notre heuristique.

La formule consiste à regarder pour chaque caractéristique s'il existe une position gagnante sur le plateau pour cette caractéristique. Si c'est le cas, on regarde le nombre de pièces qui n'ont pas cette caractéristique parmi les pièces restantes. Plus ce nombre est faible, plus on augmente la valeur de notre heuristique. Si toutes les pièces restantes ont une caractéristique qui permet de gagner avec le plateau, alors l'adversaire n'aura pas le choix et il devra nous donner la pièce qui nous fera gagner.

5 Conclusion

La réalisation de ce projet nous a permis de mettre en pratique les différents algorithmes d'intelligence artificielle vus en cours et d'être confronté aux problèmes que pose la résolution d'un jeu par l'IA. Nous nous sommes notamment heurtés à un "mur combinatoire".

Malgré les diverses difficultés que nous avons rencontré, comme par exemple la premiere modélisation du plateau, non fonctionnelle, nous sommes dans l'ensemble satisfaits des résultats obtenus.