Projet Introduction intelligence artificielle

COMPTE RENDU

Kenyu Kobayashi, Valentin Ramos

Table des matières

I. Structure implantée

a. Structure du plateau

Afin de pouvoir représenter au mieux la structure demandée, nous devions implémenter les caractéristiques suivantes :

- Un plateau de jeu
- Un joueur
- Les pions de chaque joueur
- Un plateau de liseré de jeu associé à notre plateau

Après s'être concerté, nous avons dans un premier temps, décidé de créer des classes objets correspondants à chaque pièce nécessaire à la représentation d'un plateau de jeu. Nous avions donc en ce sens, une classe Pions, Case, Plateau, Joueur. Cependant après réflexion il nous a paru plus judicieux de modifier cette structure bien que fonctionnelle. En effet, étant amené à gérer des contraintes de temps et d'espace, nous avons préféré revenir à une structure plus simple en n'ayant que pour seule classe EscampeBoard contenant :

- Un tableau de String pour chacun des deux joueurs (blanc / noir) comprenant la position de ses 6 pions.
- Un tableau 6*6 contenant les liserés de chaque case

Cette structure nous permet de générer le plateau de taille 6*6 à tout moment grâce aux deux tableaux, sans être contraint à le stocker dans un attribut. Avec cette implémentation minimale pour représenter un état de jeu, nous pouvons donc réduire le temps généré par le chargement des classes externes.

```
/** Attributs   **/
public final static char[] alphabet = {'A','B','C','D','E','F'};

// Lisere du plateau avec Lettres en j et Chiffre en i
public final static int[][] liserePlateau =
    {
        {1,2,2,3,1,2},
        {3,1,3,1,3,2},
        {2,3,1,2,1,3},
        {2,1,3,2,3,1},
        {1,3,1,3,1,2},
        {3,2,2,1,3,2}
    };

private String[] white;
private String[] black;
```

Figure 1 Extrait attributs EscampeBoard

Voici un extrait des attributs de la classe EscampeBoard. En plus du liseré et des tableaux des joueur, nous avons défini un alphabet comprenant les lettres entre A et F correspondant aux indices des tableau. C'est grâce à cet alphabet que nous pouvons facilement retrouver l'indice d'une case de type « A0 ». Nous interprétons chaque caractère afin d'obtenir l'indice i et j du plateau.

Une fonction que nous avons définie nous retournera donc par exemple 3 si la lettre est D et 0 si la lettre est A.

b. Structure d'un état

Afin de représenter un nœud dans l'arbre des coups possibles pour implémenter les algorithmes, nous avons décidé de créer une classe EtatEscampe implémentant l'interface Etat. Une instance de EtatEscampe a donc pour attributs les deux tableaux des pions joueurs, mais également le dernier coup joué pour arriver à cet état ainsi que le dernier liseré sur lequel un pion a été placé. Notons qu'il est important de stocker cette information même s'il est possible de le retrouver en connaissant la dernière case jouée : dans la cas où un joueur ne peut pas jouer, le dernier coup joué de l'état résultant serait égal à « E ». Il serait donc impossible, dans ce cas, de récupérer le dernier liseré depuis cette case.

c. Structure d'un problème

Le problème est initialisé avec l'état initial de recherche, c'est-à-dire l'état actuel. En ce sens, pour chaque coup que le joueur devra jouer, il cherchera, depuis l'état du plateau de jeu actuel, le meilleur coup en explorant les successeurs de l'état grâce à l'algorithme implémenté.

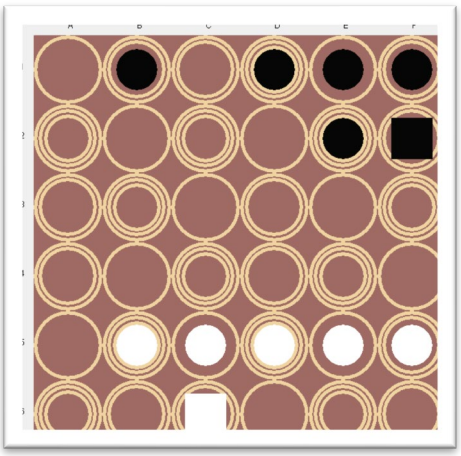
II. Meilleur coup à jouer

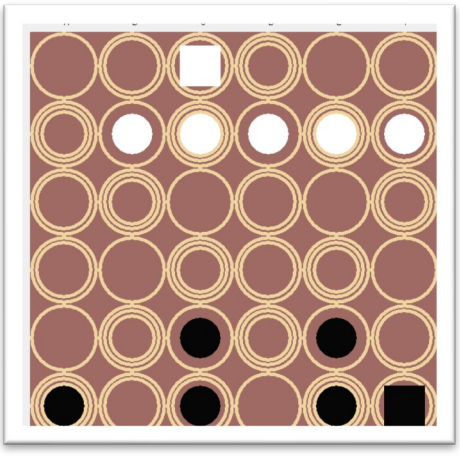
a. Placement en début de partie

Pour le placement en début de partie, il nous a semblé judicieux de définir deux placements optimaux dans toutes circonstances. Les critères sont les suivants :

- Protéger la licorne
- Positionner les paladins en formation offensive
- Répartir suffisamment de pions sur chaque liseré

Tenu compte de ces critères, en voici les deux résultats de placements :





Dans les deux cas, la licorne est à l'arrière, protégée par ses paladins qui sont alignés devant.
Par ailleurs, mettons l'accent sur le fait que cette disposition permet d'avoir deux pions sur chaque liseré, permettant ainsi une flexibilité dans les choix de mouvements en début de partie.

b. Meilleur coup en milieu de partie

Le meilleur coup en milieu de partie est estimé par notre algorithme de recherche Alpha Beta couplé à notre heuristique.

III. Choix des heuristiques

Durant ce projet, plusieurs heuristiques, reposants chacune sur différents critères ont été implémentées :

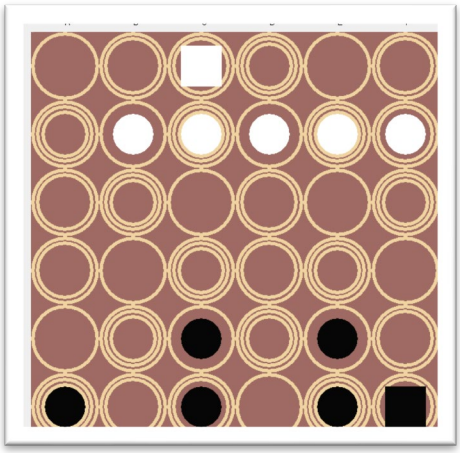
- La première heuristique, hEnd, renvoie moins l'infini lorsqu'un état correspond à la défaite d'un joueur ami, et plus l'infini lorsqu'il correspond à la victoire d'un joueur ami.

```
//Heuristique hEnd
public float evalEnd(Etat e) {
    if (e instanceof EtatEscampe) {
        EtatEscampe ee = (EtatEscampe) e;
        String[] pions_ami, pions_ennemi;

        //Cas ou le joueur ami est blanc
        if(this.player.contains("blanc") ) {
            pions_ami = ee.getWhite();
            pions_ennemi = ee.getBlack();
        }
        //Cas ou le joueur ami est noir
        else {
            pions_ennemi = ee.getWhite();
            pions_ami = ee.getBlack();
        }

        //Si la licorne ami est morte
        if ( pions_ami[0].contains("ZZ") ) {
            return Integer.MIN_VALUE;
        }
        if ( pions_ennemi[0].contains("ZZ") ) {
            return Integer.MAX_VALUE;
        }
        return 0;
    } else {
        throw new Error("Cette heursitique ne peut s'appliquer que sur des EtatEscampe");
    }
}
```

- La deuxième heuristique, hLis, étudie les liserés des cases sur lesquelles les pions sont positionnés. Pour chaque type de liseré différent que le joueur ami possède, on ajoute 20 à la valeur de retour, qui est 0 au départ.
  Par ailleurs, pour chaque type de liseré différent que le joueur ennemi possède, on retire 20 à la valeur de retour.

Cette heuristique favorise la disposition de toujours une pièce sur chaque type de liseré, pour éviter la situation ou le joueur est obligé de jouer.

Exemple :

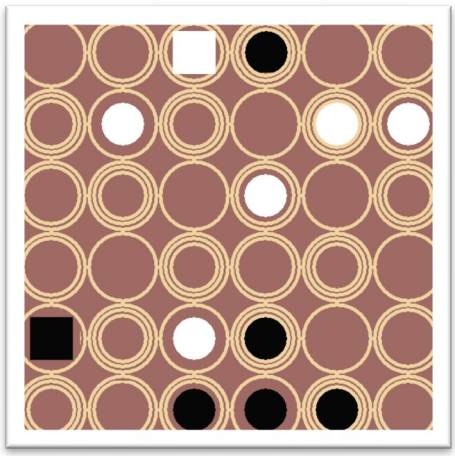


Soit res la valeur de retour de l'heuristique. On part de res = 0.
Dans cette situation, le joueur blanc (ami) possède des pions sur chaque type de liseré.
On a donc res = res + 20*3
Cependant, le joueur noir (ennemi) possède également des pions sur chaque type de liseré.
On a donc res = res -20*3.
Finalement, la valeur de retour res = 0.

- La troisième heuristique, hDist, consiste à étudier les distances qui séparent les licornes des paladins. La licorne amie doit éviter de se faire prendre par les paladins ennemis, et les paladins amis ont pour but de prendre la licorne ennemie.
  Nous avons donc calculé l'heuristique de la manière suivante :
  - Soit **dist_max** la somme des distances séparant la licorne amie aux paladins ennemis. Cette distance doit être alors maximale pour que le joueur ami soit à l'avantage.
  - Soit **dist_min** la somme des distances séparant la licorne ennemie aux paladins amis. Parallèlement, cette distance doit être alors minimale pour que le joueur ami soit à l'avantage.
  - Dans le cas où un paladin est à une distance inférieure à 3 de la licorne, nous estimons qu'il représente une menace et lui attribuons un **poids 2 fois supérieur aux autres.**

  Finalement, l'heuristique hDist renvoie donc à l'issue de ces calculs **dist_max – dist_min**.

  Exemple :



Soit res la valeur de retour de l'heuristique.
Le joueur blanc est l'ami.
dist_max = 5+6+7+6+ 1/2 = 24.5
dist_min = 4+7+8+5+ 2/2 = 25
res = dist_max – dist_min = -0.5
Cette situation n'est pas très avantageuse pour le joueur ami.

- La dernière heuristique, hFinal, consiste à regrouper les critères des trois premières heuristiques, en plus d'un autre critère reposant sur les liserés.
  Ce critère utilise les distances séparant la licorne aux paladins, calculés dans hFinal.
  Si un paladin ennemi est à une distance inférieure à 3 de la licorne amie, alors on regarde la situation de plus près en faisant plusieurs tests :

  - Test 1 : Soit le liseré sur lequel est le paladin ennemi concerné. Si ce liseré est égal à la distance qui sépare la licorne à ce paladin, et que la licorne ne peut pas bouger ce tour, la situation est **inquiétante**. Appelons ce liseré LMort et le paladin PMort. On passe au test 2.
  - Test 2 : Regardons s'il existe un chemin valide (i.e. sans obstacles) entre PMort et la licorne amie. S'il en existe un, la situation est **grave**. Passons au test 3.
  - Test 3 : Regardons s'il existe un pion ami capable de se déplacer sur une case dont le liseré est différent de LMort. S'il n'en existe pas, la partie est **perdue**.
    L'heuristique renvoie donc moins l'infini.

  Si la situation est **grave**, on soustrait 100 à la valeur renvoyée par l'heuristique.
  Si la situation est **inquiétante**, on soustrait 50 à la valeur renvoyée par l'heuristique.
  Si aucuns de ces tests ne sont validés, la valeur renvoyée par l'heuristique sera simplement celle selon hDist. C'est pourquoi, soustraire 100 ou 50 à ce résultat a un impact conséquent sur la valeur de retour.
  De manière analogue, on procède aux mêmes tests si jamais nous avons un paladin ami à une distance inférieure à 3 de la licorne ennemie.
  La situation pourrait être **favorable**, voir très **favorable** pour le joueur ami selon la validation des différents tests.

IV. Gestion du temps réel

a. Solution mises en œuvres

Afin de respecter la contrainte de temps imposée par le jeu pour chaque partie et maximiser le score, nous avons décidé d'utiliser l'élagage Alpha Beta en couplé à la politique de recherche à profondeur incrémentale « Iterative Deepening ».
L'algorithme Alpha Beta permet de réduire le nombre de nœuds évalués par l'algorithme miniMax, permettant ainsi un gain de temps.
Par ailleurs, Iterative Deepening nous permet d'explorer un maximum de nœuds tout en respectant une contrainte de temps. En effet, en implémentant un minuteur dans Alpha Beta, nous pouvons ajouter une condition d'arrêt à notre algorithme : il nous suffit simplement de vérifier, avant chaque itération de l'algorithme, si la durée d'exécution depuis le début de l'appel est supérieure au temps maximal imposé.
Si ce cas est vérifié, nous interromprons l'algorithme en renvoyant la valeur du coup estimé comme étant le meilleur jusqu'à là.
Pour finir, nous avons privilégié les états successeurs avantageux en début de liste, pour maximiser le gain de temps de l'élagage Alpha Beta.

```
//On recupere les mouvements qui prennent la licorne ennemie pour les placer au debut du tableau
if ( (cases.contains(white[0]))||(cases.contains(black[0])) ) {
    possible_moves.addFirst(move);
}
else {
    possible_moves.add(move);
}
```

V. Analyse des performances

a. Test des différents algorithmes et heuristiques

Après avoir implémenté les différentes heuristiques, nous avons testé ces dernières dans des conditions réelles. Nous avons pu, en conséquence, construire un tableau comparatif de nos différentes observations.

Heuristique/Critères	Temps d'exécution moyen(ms)	Profondeur moyenne	Score contre joueur moyen
hDist	190	8	1/5
hLis	280	8	1/5
hEnd	10	12	4/5
hFinale	9	8	5/5

b. Choix et explication

Lors de la comparaison des heuristiques, nous avons pu observer que hFinal est l'heuristique la plus performante. Cependant, ce n'est pas une surprise. Les heuristiques hLis et hDist manquaient tous deux de hEnd, l'heuristique qui permet d'éviter la défaite et de favoriser la victoire.
Bien que classique, hEnd tient un bon score. Non seulement elle a un temps d'exécution largement inférieur à hDist et hLis, elle permet aussi une exploration beaucoup plus profonde.
C'est donc hFinale, qui fusionne ces 3 heuristiques, qui l'emporte avec un score de 100% contre un joueur moyen.
Nous ne sommes pas sûrs quant à la vitesse d'exécution de l'algorithme couplé a cette heuristique, sûrement dû au fait qu'il trouve rapidement un chemin vers un état terminal.
Observons également que le temps de recherche augmente de façon significatives dans un cas particulier : Si le joueur peut jouer plus de 2 pions sur un liseré 3 et que leurs voies sont dégagées, alors le temps sera grandement augmenter. Ceci est explicable du fait que pour chaque coup possible, on augmente le nombre de branches à explorer. Le liseré 3 propose 12 coups soit 36 coups pour 3 pions en conséquence 36 branches fils pour l'état initial.

VI. Tests effectués

Pour mener notre projet à bien, nous avons coder un affichage graphique rapide lors de la simulation de parties solos pour débugger rapidement :

```
*********
On demande à Kobayashi Kenyu - Ramos Valentin de jouer...
Coups possibles de blanc:
E2-F4,E2-D4,E2-C3,
Nombre de feuilles développés par la recherche : 3
Nombre de noeuds développés par la recherche : 2
Temps de la recherche: 2 ms.
Profondeur: 2
Temps moyen: 10, profondeurMoyenne: 9
Meilleur Coup : E2-D4
Le joueur Kobayashi Kenyu - Ramos Valentin a joué le coup E2-D4 en 2ms.
- - B - - -     1|2|2|3|1|2|
- b - b - b     3|1|3|1|3|2|
- - - - - -     2|3|1|2|1|3|
- b n b - -     2|1|3|2|3|1|
- n - n n n     1|3|1|3|1|2|
- - N - - -     3|2|2|1|3|2|

*********
```

Figure 2 Execution courte

Dans cette exécution, nous pouvons observer toutes les informations nécessaires à l'évaluation de l'heuristique : en effet, nous pouvons y retrouver le temps moyen ainsi que la profondeur moyenne explorée par l'algorithme.
Afin de pouvoir observer les fonctionnalités de l'algorithme, nous déciderons par la suite de descendre le temps maximal de 10 à 6 secondes. Ceci nous permettant d'obtenir le résultat suivant :

```
*********
On demande à Kobayashi Kenyu - Ramos Valentin de jouer...
Coups possibles de noir:
C1-C3,C1-D2,C1-B2,C1-E1,C1-A1,A3-A1,A3-B2,A3-A5,A3-C3,D4-D6,D4-F4,D4-C3,D3-D1,D3-C2,D3-C4,D3-B3,F2-E1,
Meilleur Coup : C1-A1
Le joueur Kobayashi Kenyu - Ramos Valentin a joué le coup C1-A1 en 6003ms.
Time over, waitedTime: 6003 ms.
N - - - - -     1|2|2|3|1|2|
- - - - n n     3|1|3|1|3|2|
n - - n b b     2|3|1|2|1|3|
- b - n - -     2|1|3|2|3|1|
- - b - b -     1|3|1|3|1|2|
- - - - B -     3|2|2|1|3|2|

*********
```

Nous avons dans le cas ci-dessus, un nombre trop conséquent de nœuds à explorer ne permettant pas le développement de tous ces derniers dans le temps imposé. Nous pouvons observer en rouge l'interruption de l'algorithme pour cause de non-respect de la contrainte de temps imposée à 6 secondes. Cette exécution montre bien la différence de complexité entre deux états suivant le positionnement des pions ainsi que du liseré précédent.

VII. Conclusion et difficultés rencontrées

En conclusion, ce projet a été une première pour nous dans le domaine de l'intelligence artificielle, et a été très intéressant à mener. L'idée de faire affronter plusieurs intelligences artificielles entre camarades de classe a été une source de motivation très appréciée.
Ce projet nous a permis d'appliquer les algorithmes vus en cours, qui pour la plupart n'étaient pas parfaitement assimilés.
Cependant, nous avons malheureusement sous-estimé la deuxième partie du projet, plus particulièrement dans la recherche des coups possibles en respectant les règles imposées, ce qui explique en partie notre retard.
Le projet nous aura toutefois beaucoup enseigné, notamment dans l'utilisation du plugin de debug qui nous a été grandement utile durant tout le projet.
Pour finir, nous sommes plus que satisfaits du résultat final.