Développement d'une intelligence artificielle pour un jeu à 2 joueurs - Documentation technique

Sylvain BESNARD, Romain LE BORGNE,
Fabien BRAULT, Baptiste BIGNON

Encadreur : Christian RAYMOND

1 Introduction

Nous allons présenter de manière succincte les parties importantes du code de notre projet. Nous allons tout d'abord présenter l'implémentation du jeu de Dames, puis l'implémentation graphique et enfin l'implémentation de l'intelligence artificielle.

Nous avons également essayé tout au long du développement de commenter au maximum notre code afin qu'il soit le plus clair possible.

2 L'implémentation du jeu de Dames

Nous avons réalisé l'implémentation du jeu en lui-même dans les classes Game, Checkerboard et Player.

2.1 La classe Game

La classe Game correspond à la création d'une partie selon les paramètres lui étant indiqués (taille du plateau, mode et niveau des joueurs, etc). Elle contient donc deux joueurs ainsi que le plateau de jeu. Elle permet de switcher entre les joueurs et de gérer le bon déroulement de la partie.

2.1.1 Gestion du déroulement de la partie

Le déroulement de la partie est défini grâce à une machine à état. La classe contient un attribut state pouvant prendre les valeurs indiquées dans l'enum STATE du fichier struct.h.

Voici le déroulement du jeu selon notre machine à états :

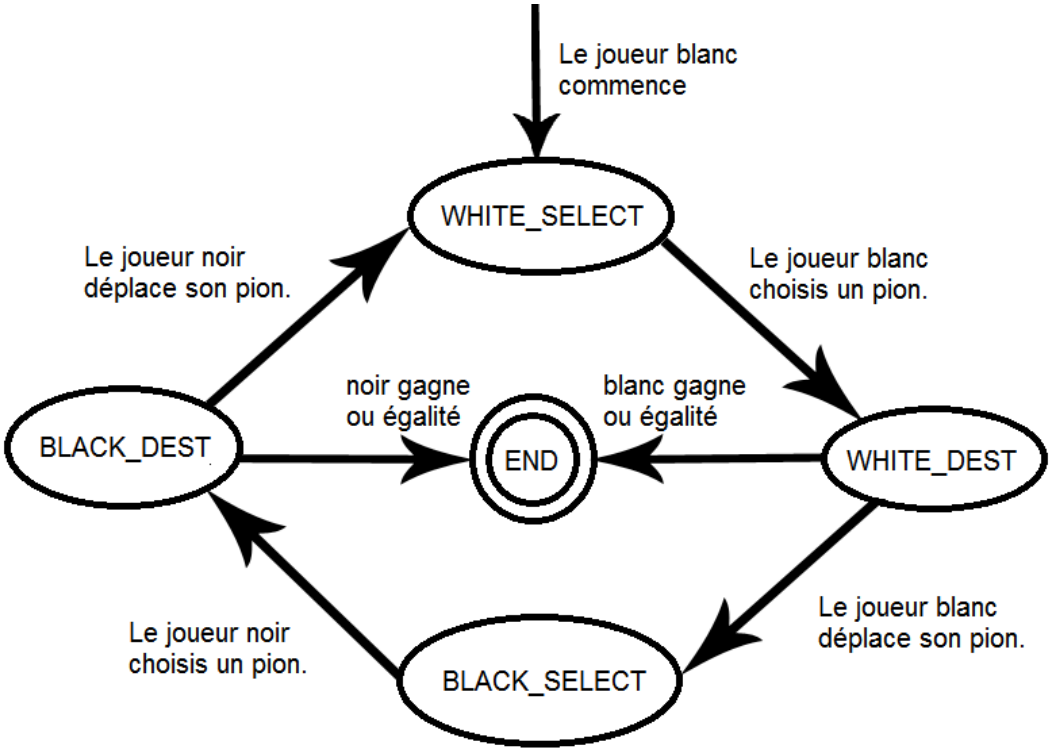

FIGURE 1 – Schéma du déroulement du jeu

2.1.2 Les fonctions de l'intelligence artificielle

La classe Game contient l'ensemble des fonctions nécessaires au calcul des choix des joueurs non manuel (Partie 4 - L'implémentation de l'intelligence artificielle).

2.1.3 La gestion du reporting

L'initialisation, la création et l'enregistrement du reporting à chaque coup est implémenté dans cette classe.

2.2 La classe Checkerboard

La classe Checkerboard contient les cases du plateau de jeu. Ces cases sont de type struct QSQUARE défini dans le fichier struct.h.

Elle comporte aussi des méthodes applicables sur ce plateau. Chacune d'entre elles sont commentées.

2.3 La classe Player

La classe Player contient les informations propres à un joueur (level, fonction de coût associée, son pion, sa dame, son état SELECT, son état DEST, etc). Les attributs x, y, xDest et yDest correspondent aux coordonnées du dernier mouvement du joueur (ou du mouvement qu'il est en train de réaliser). Les coordonnées (x,y) définissent son pion sélectionné et les coordonnées (xDest,yDest) la case de déplacement choisie.

La classe Player comportent la majorité des fonctions concernant le déplacement d'un pion sur un plateau de jeux, du fait que le déplacement d'un pion dépend du joueur : chaque joueur ne peux déplacer un pion que vers la ligne ennemie.

3 L'implémentation graphique

L'implémentation graphique du jeu a été réalisée en utilisant le framework Qt. Afin de faciliter la mise en place de l'interface nous avons utilisé Qt Designer (voir image ci-dessous) pour positionner les éléments de base.

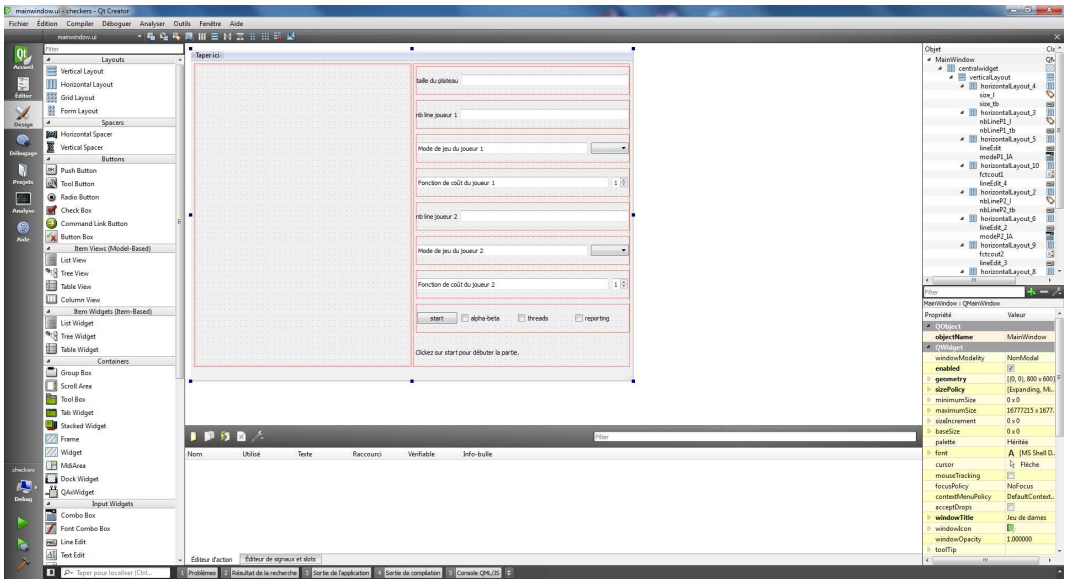

FIGURE 2 – Interface de Qt Creator

Ceci est complété par la classe MainWindow (qui hérite de QMainWindow) qui met en place les différents slots et signaux permettant d'interagir avec le jeu via l'interface graphique, ainsi que les différentes fonctions nécessaires à l'affichage du jeu. Vous pouvez vous reporter aux commentaires présents dans mainwindow.h pour connaître les différentes fonctions existantes et leur utilité.

La classe LabelCase qui hérite de QLabel permet de gérer des labels contenant leur position et pouvant contenir et afficher une image. Ces labels représentent les cases du plateau.

4 L'implémentation de l'intelligence artificielle

L'intelligence artificelle a été implémentée dans la classe Game par les fonctions negaMax et alpha-beta.

4.1 Les fonctions d'appel

```
MOVE negaMax(bool with_thread_param)  ;
MOVE alphaBeta(bool with_thread_param) ;
```

Ces deux fonctions initialisent les paramètres nécessaires et lance l'exécution respective des algorithmes MegaMax et AlphaBeta dans l'ordre suivant :

- Initialisation du reporting
- Initialisation du temps de début d'exécution de l'algorithme
- Recherche d'un pion en cours de sélection sur le plateau, limitant la recherche de mouvement à ceux partant de ce pion
- Lancement de l'algorithme avec les bons paramètres
- Sauvegarde du reporting
- Calcul du temps d'exécution total
- Renvoie une erreur si aucun mouvement n'a été trouvé
- Renvoie un mouvement au hasard parmi les meilleurs mouvements

4.2 Les fonctions récursives

Il y a 4 fonctions récursives :

```
int negaMaxClassic(const Checkerboard & board, int depth, COLOR color, Player* P1,
Player* P2, std::vector<MOVE> & best, int xSelect, int ySelect) ;

int negaMaxThread(const Checkerboard & board, int depth, COLOR color, Player* P1,
Player* P2, std::vector<MOVE> & best, int xSelect, int ySelect) ;

int alphaBetaClassic(const Checkerboard & board, int depth, COLOR color, Player* P1,
Player* P2, std::vector<MOVE> & best, int maxprec, bool ismaxprec, int xSelect,
int ySelect);

int alphaBetaThread(const Checkerboard & board, int depth, COLOR color, Player* P1,
Player* P2, std::vector<MOVE> & best, int maxprec, bool ismaxprec, int xSelect,
int ySelect);
```

Ces fonctions prennent en paramètres :

- Le plateau de jeux du nœud actuel dans l'arbre de l'algorithme
- Le niveau de profondeur actuel
- La couleur du joueur dont le mouvement est simulé dans le niveau actuel
- Les joueurs de la partie
- La liste des meilleurs mouvements trouvés actuellement
- Les coordonnées (xSelect, ySelect) de la case sélectionnée si le coup en cours de calcul est un coup à plusieurs prises : le joueur ne peut pas désélectionner son pion et le calcul des mouvement doit se faire depuis ce pion

Fonctionnement :

- Teste si le nœud actuel est une feuille. Le nœud actuel est une feuille dans 2 cas :
  - Il y a un gagnant ou une égalité sur le nœud actuel
  - Le niveau de profondeur est celui indiqué comme maximal par le niveau de l'IA choisie
- Dans le cas où le nœud est une feuille, la fonction retourne la valeur retournée par l'appel à la fonction de coût sur le plateau feuille
- Sinon, l'algorithme poursuit avec la recherche de tout les nœuds fils via la fonction « findChild »
- Un appel récursif à l'algorithme est appliqué à chaque nœud fils
- La fonction retourne la meilleure valeur de fonction coût issue de l'appel récursif de tout ses fils

4.2.1 Les particularités de l'AlphaBeta :

Les fonctions AlphaBeta prennent 2 paramètres supplémentaires :

- maxprec : il s'agit de la meilleure valeur de fonction de coût actuellement trouvées parmi les frères du nœud père du nœud actuel
- Ismaxprec : il s'agit d'un booléen indiquant si la valeur maxprec est à prendre en compte, celle-ci n'est pas à prendre en compte lorsque le nœud père est le premier fils de ses frères car il n'y a alors pas encore de valeur maximal vu que que la première est en cours de calcul via le nœud actuel

Elles ont de plus une variable locale *nb_ child_ treated* qui a pour valeur le nombre de fils du nœud actuel sauf si une coupure Alpha-Beta a pu être réalisée. Le test de réalisation possible d'une coupure AlphaBeta se situe à la condition suivante :

```
if (i!=0 && ismaxprec && value>maxprec && nb\_child\_treated==child.size());
```

Une coupure AlphaBeta est possible si :

- Il ne s'agit pas du premier fils à étudier
- La valeur « maxprec » est à prendre en compte (le nœud père n'est pas le premier fils de son père)
- La valeur actuelle est supérieure à la valeur actuellement trouvée parmi les frères de son père (ce qui signifie qu'une fois remontée au niveau de profondeur supérieur, elle sera inférieure ou égal à la valeur actuelle parmi les frères, voir fonctionnement de l'AlphaBeta dans le compte rendu)

4.2.2 Les particularités des threads

- L'appel initial aux fonctions récursives est implémenté de la manière suivante :

```
#pragma omp parallel
{
#pragma omp single
Appel à la fonction récursive de l'algorithme
}
```

  - *#pragma omp parallel* encadre l'ensemble de l'algorithme
  - *#pragma omp single* indique que l'on souhaite, pour le moment, que l'algorithme soit exécuté via un seul thread
- Au sein des fonctions récursives se trouvent les indications suivantes :

```
for each childs
{
...
#pragma omp task shared(best)
Appel récursif
}
#pragma omp taskwait
```

  - *#pragma omp task shared(best)* indique le début d'une tâche exécutable par un autre thread, l'appel récursif de la fonction sur chaque fils pourra donc être exécuté par d'autres threads. *shared(best)* permet de partager le tableaux des meilleurs coups entre tout les threads pour que chacun puisse le mettre à jour

  - *#pragma omp taskwait* indique l'attente de tous les threads lancés précédemment avant de poursuivre l'exécution du code en ce point

- Les points critiques :
  - *#pragma omp critical* : des blocs de code ont été protégés via cette indication, il s'agit des instructions modifiant la variable best, partagés entre chaque thread comme indiqué précédemment. Ainsi, pour éviter les conflits, un seul thread à la fois pourra modifier ce tableau. Une autre protection a été ajoutée pour protéger l'affectation de la meilleure valeur actuelle, nécessaire d'être modifiée au fur et à mesure par chaque thread dans le cas d'un algorithme AlphaBeta pouvant effectuer une coupure selon cette valeur en cours de boucle