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 :

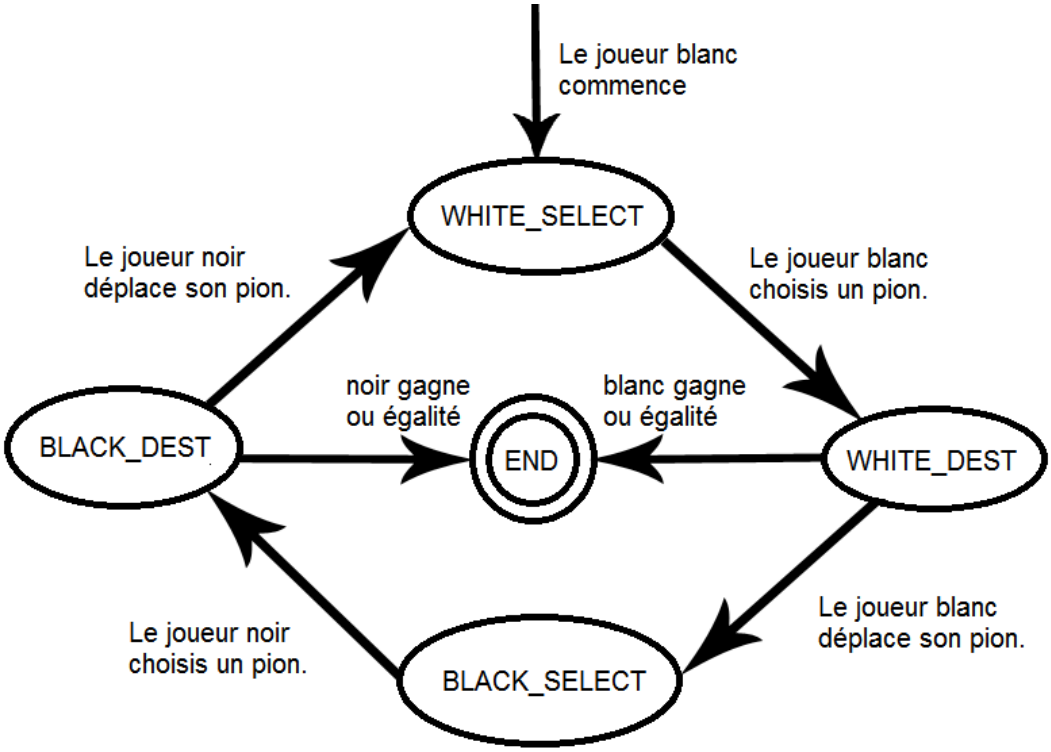

FIGURE 1 – Schéma du déroulement du jeu

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.

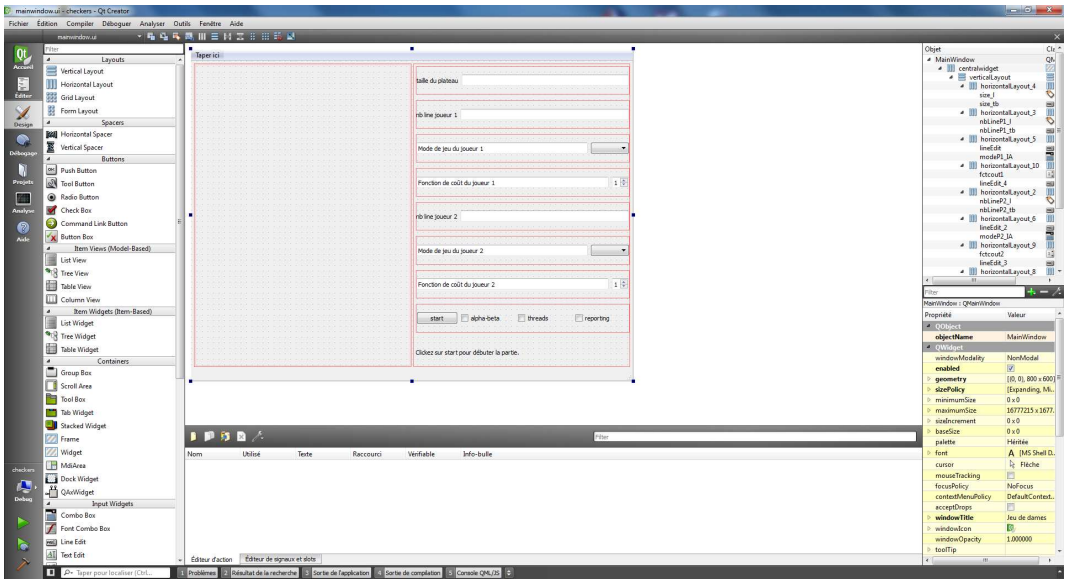

FIGURE 2 – Interface de Qt Creator

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

```
int Game::negaMaxClassic(const Checkerboard & board, int depth, COLOR color,
Player* P1, Player* P2, std::vector<MOVE> & best, int xSelect, int ySelect) {
    // init the current player and opponent simulation
    Player* player = (color==WHITE ? P1 : P2) ;
    Player* opponent = (player==P1 ? P2 : P1) ;
    int value ;
    std::vector<CHILD> child ;
    child.clear() ;
    // cost function return if the depth is max are if the game is finish on the
current board
    if (depth==0 || isFinishOnBoard(board, player)){
```

```
        value = costFunction(board, color) ;
        // reporting operations if necessary
        if(with_reporting) {
            timeval end ;
            gettimeofday(&end , NULL) ;
            add_node_reporting(board,value,Tools::timediff(time_IA_begin,end),
child.size(),child.size()) ;
        }
        return value ;
    }
    // find all the child corresponding with the current node board
    child = findChild(board,color, player, xSelect, ySelect) ;
    // for each childs
    for (int i = 0 ; i<(int)child.size() ; i++) {
        // keep the graphism active
        QCoreApplication::processEvents();
        if (_stop) exit(EXIT_SUCCESS);
        // indicate the threads informations
        if (omp_get_num_threads()>1)
            qDebug() << "nb threads = " << omp_get_num_threads() ;
        // launch algorithm recursively and save cost value
        child[i].value = - negaMaxClassic(child[i].board, depth - 1, (COLOR)
(-(int)color),P1, P2, best, child[i].xSelect, child[i].ySelect) ;
    }
    // find the best child(s) and the best move(s) corresponding
    value = findBestChild(child,best,child.size()) ;
    // reporting operations if necessary
    if(with_reporting) {
        timeval end ;
        gettimeofday(&end , NULL) ;
        add_node_reporting(board,value,Tools::timediff(time_IA_begin,end),
child.size(),child.size()) ;
    }
    return value;
}
```