Bruno Cornil, Matricule : 000392857

Rapport INFO-H-304

Description générale du programme:

La structure du jeu est illustrée dans la figure 1 ci-dessous :

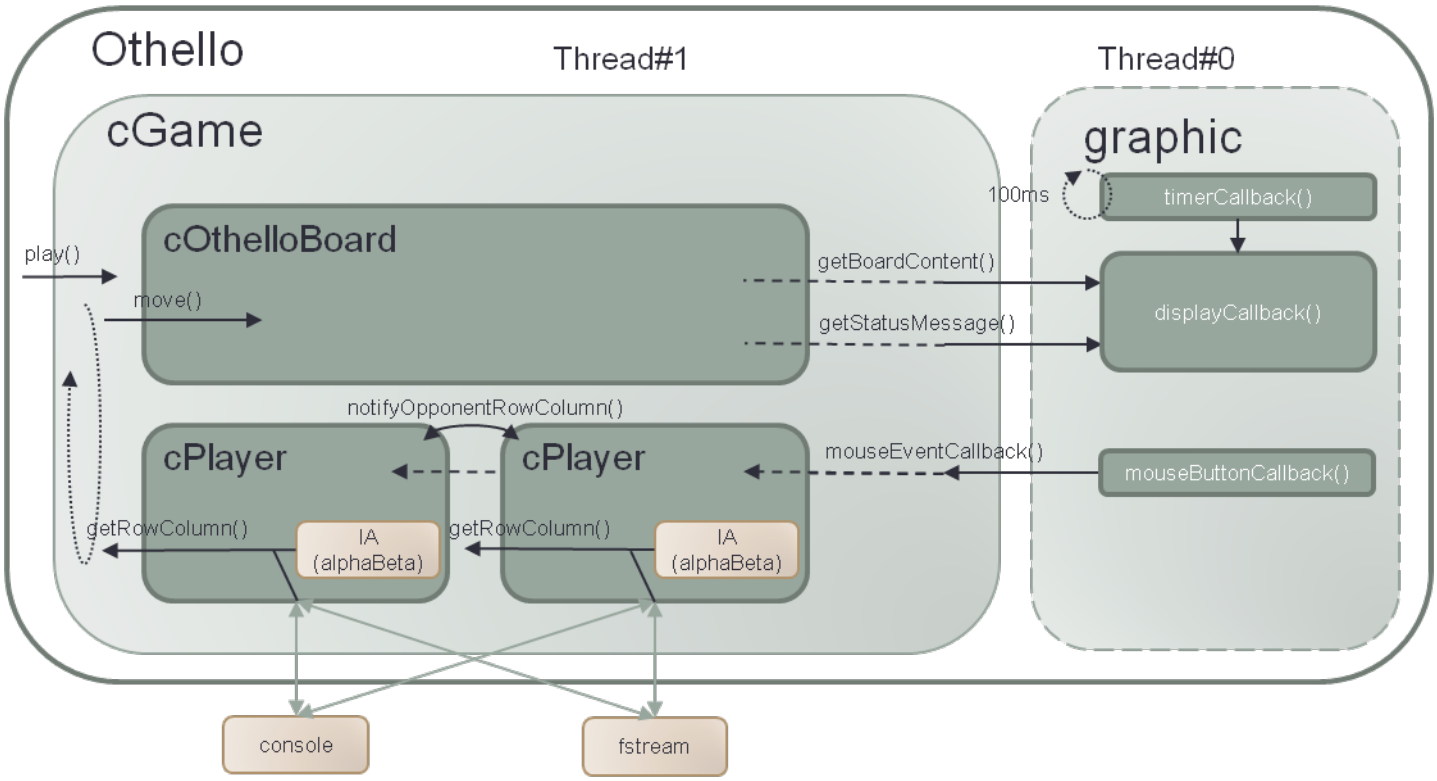

Figure 1 : Structure du programme

Le programme démarre dans othello.cpp en créant un object *cGame* puis en appelant sa méthode *mainLoop()* qui ne retourne qu'une fois le jeu terminé. Si le jeu a été compilé avec une interface graphique, la fonction *mainLoop()* de l'objet *cGame* est démarré dans une nouvelle thread car il est plus simple d'utiliser la thread principale pour l'interface graphique (c'est elle qui reçoit tous les évènements (souris, fenêtres, ...).

Les paragraphes suivant donnent une description des classes et fonctions principales du programme et illustrées ci-dessus. Une documentation succincte de chaque fonction est aussi inclue dans le code source.

Cgame:

cGame implémente le jeu; il crée un plateau de jeu (*board*, classe *cOthelloBoard*) ainsi que deux joueurs de couleur différente (*players*[2], classe *cPlayers*).

La méthode *mainLoop*() de la classe appelle la fonction *play*() tant que le jeu n'est pas terminé (fonction *isGameOver*()) puis affiche le résultat de la partie (fonction *displayScore*())

Toutes les combinaisons de joueurs sont possibles, sauf bien sur deux joueurs de type (F), on peut par exemple démarrer un jeu avec un jouer de type (A) et de type (F) qui communiquent avec un autre jeu configuré pour des joueurs de type (F) et (H) par l'intermédiaire de fichiers d'échange

(A) contre (F) □ fichiers □ (F) contre (H)

La méthode *play*() commence par une mise à jour du message d'information de l'objet *board* (qui sera affiché soit sur la console soit dans la fenêtre graphique), affiche le tableau de jeu (*cBoard::display()*) puis, si un mouvement est possible (*cBoard::isMovePossible()*) demande au joueur courant un nouveau coup (*cPlayer::getRowColumn*()), communique ce coup a l'adversaire (nécessaire uniquement s'il est de type (F)) pour ensuite passer la main au joueur suivant.

Si l'interface graphique est utilisée, la classe *cGame* implémente trois méthodes supplémentaires pour interagir avec celui-ci (l'environnement graphique ne connait pas les objets *board* et *players* créé par cGame, et ce dernier agit donc comme intermédiaire pour passer ou fournir des informations vers/de ceux-ci).

- *mouseEventCallback*: permets de passer au joueur courant, la rangée et colonne suite à un clic de la souris dans la fenêtre (remplace l'entrée du coup via le clavier).

- *getBoardContent* et *getStatusMessage*: permettent à la fonction d'affichage graphique de lire le contenu du plateau de jeu et des messages d'information.

Cplayer:

Lors de la création d'un objet *cPlayer*, l'utilisateur doit spécifier si le joueur est de type humain (H), fichier (F) ou bien une intelligence artificielle (A).

Un joueur (H) utilise la console pour communiquer avec l'utilisateur, un joueur (F) utilise un fichier par direction (l'utilisateur doit alors spécifier le répertoire utilisé par les fichiers d'échange) similaire à l'exemple du TP 'sudoku' tandis que pour le joueur (A) il agit de manière autonome.

La méthode getRowColumn() est la fonction principale de l'objet cPlayer et collecte la rangée et colonne d'un nouveau coup en fonction du type de joueur.

Si le joueur est une intelligence artificielle, le joueur calcule lui-même la position optimale sur base d'un algorithme « AlphaBeta » implémenté dans la fonction ... *alphaBeta*().

L'algorithme « AlphaBeta » est souvent utilisé pour implémenter une IA avec un tableau de jeu et dérive de l'algorithme « MinMax ». Ce dernier essaie toute les positions possible (et valide) de manière récursive jusqu'une profondeur donnée et calcule une figure de mérite (vue du point de vue de l'AI). Ce score est alors utilisé par le niveau précèdent pour choisir le meilleur coup. Si ce niveau est celui de l'adversaire la valeur minimale est logiquement choisie (il veut faire perdre l'AI), s'il s'agit du niveau de l'AI la valeur maximale est choisie (l'AI veut gagner) ... d'où le nom « MinMax ».

Afin d'optimiser le temps de recherche l'algorithme « AlphaBeta » va diminuer le nombre de branches à calculer par le MinMax en se basant sur les observations suivantes:

- l'adversaire essaie de minimiser le score de l'IA et ne choisira donc pas un score supérieur au score maximal déjà en sa possession => si ce dernier est communiqué au niveau Max/de l'AI (paramètre « Beta ») et que l'AI découvre une option avec score supérieur (et que l'AI va donc choisir) on abandonne la recherche (« cassure Beta » dans l'arbre) car l'adversaire choisira de toute façon l'option inférieure (« Beta ») pour essayer de gagner.
- De manière similaire, si le score inférieur de l'AI est communiqué au niveau Min/de l'adversaire, celui-ci abandonnera (« cassure Alpha ») s'il trouve un score encore plus bas (qu'il va choisir mais l'AI choisira elle au moins « Alpha »).

Un exemple est illustré dans la figure 2 ci-dessous :

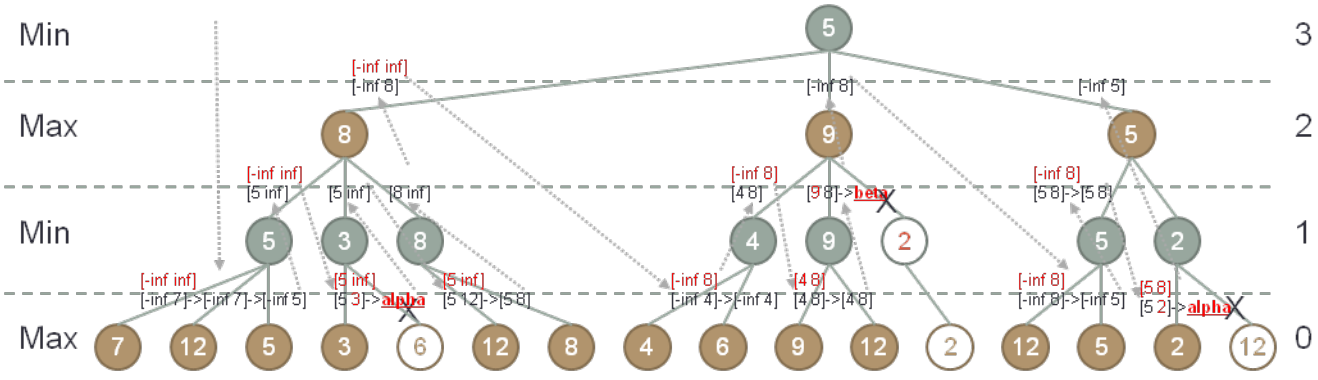

Figure 2 : Arbre MinMax et coupure AlphaBeta

La figure 3 compare l'évolution du nombre de nœuds analysés par l'algorithme « MinMax » et « AlphaBeta » pour les dix premiers coups du jeu (deux joueurs IA) et pour différentes profondeurs d'analyse. On observe que l'algorithme AlphaBeta diminue considérablement le nombre de recherche ($\sim 10^{0.5}$ soit un facteur 3 ou approximativement un niveau supplémentaire pour le même coût) et a donc été choisi pour ce projet.

La figure 4 illustre l'évolution de la profondeur d'analyse pour un jeu complet. On peut observer que le nombre de nœuds à analyser est plus élevé dans la première moitié de la partie (normal puisque la grille est bien ouverte), diminue d'un facteur dix dans la deuxième partie (on pourrait donc augmenter la profondeur d'analyse de un niveau pour un coût similaire) et juste rapidement pour les dix derniers coups (on pourrait changer d'algorithme et augmenter la complexité de la fonction d'évaluation du score)

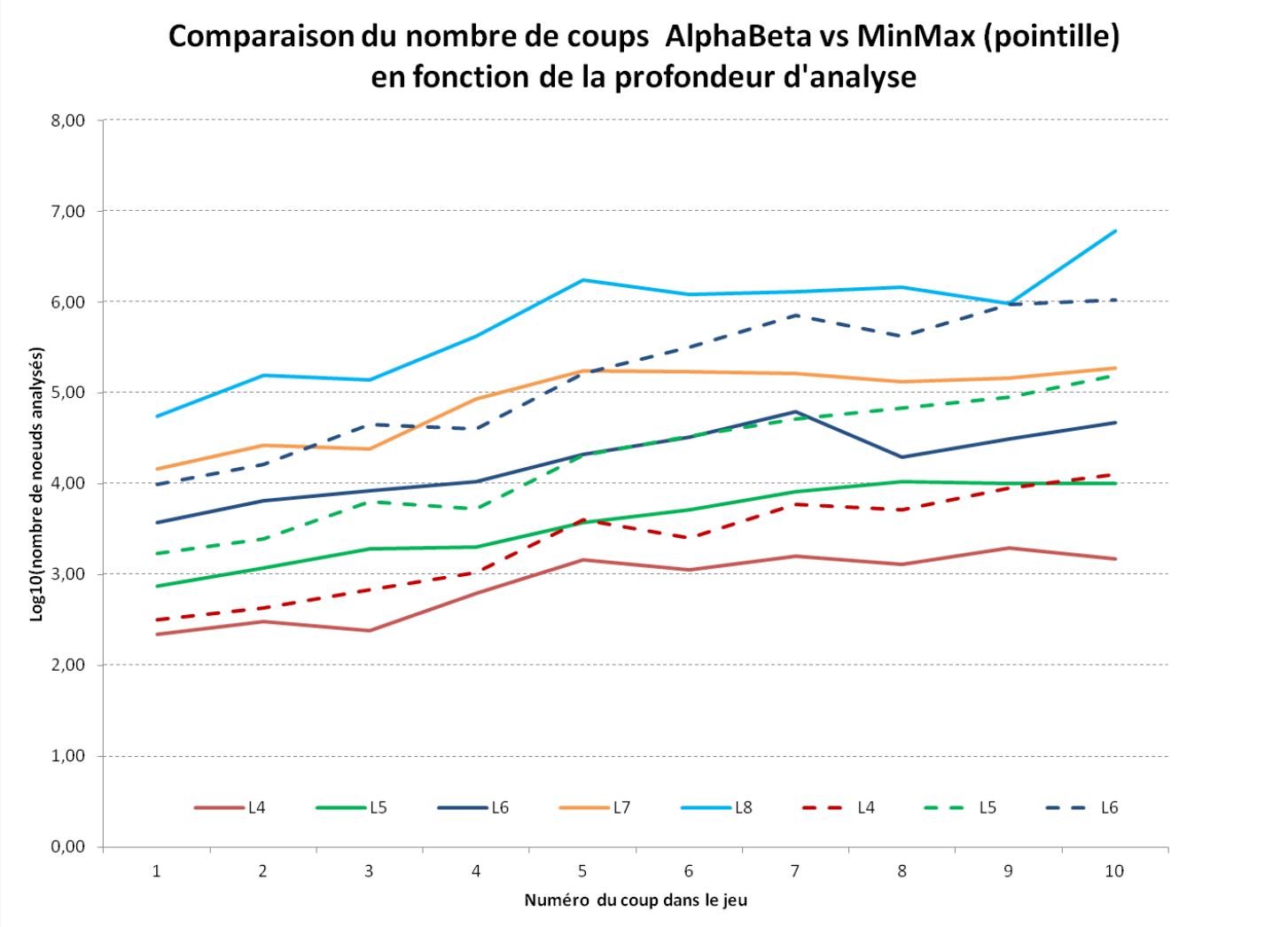

Figure 3 : Nombre de nœuds calcule pour les algorithmes MinMax et AlphaBeta

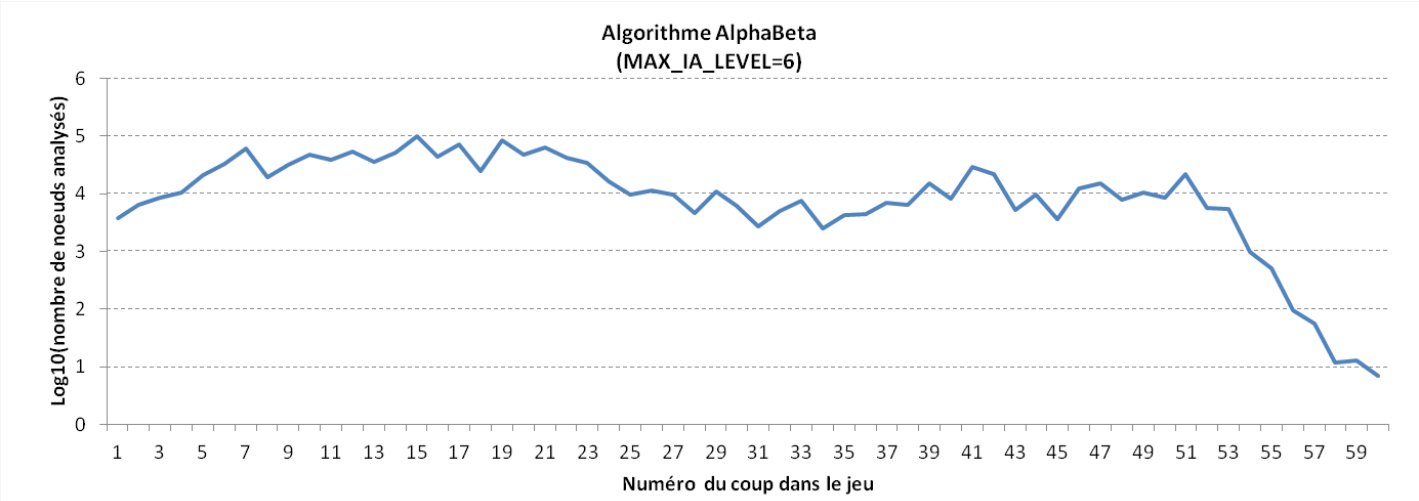

Figure 4 : Evolution du nombre de nœuds pour l'algorithme AlphaBeta

Le choix d'une fonction score n'est pas trivial pour un jeu comme Othello, j'ai choisi d'utiliser le nombre d'ouverture possible (vu de l'IA). Je pense que c'est une bonne indication du niveau de contrôle qu'a un joueur sur le jeu lorsque celui-ci est encore 'ouvert'. Il faudrait probablement considérer le nombre de pions immuables ainsi que les positions stratégiques (ex. la valeur d'un coin) ou instables (ex. les lignes et colonnes distance d'une case du bord du plateau) pour améliorer l'algorithme, et ce certainement vers la fin du jeu ... et avec plus de temps.

Pour ajouter une couleur «humaine»/imprévisible à l'algorithme, on pourrait également choisir de manière aléatoire dans la liste des scores qui sont équivalents plutôt que de toujours conserver le premier.

Le programme n'implémente également pas de limitation stricte du temps de recherche mais limite la profondeur de recherche pour limiter le temps pris par l'IA (celle-ci est spécifiée dans le fichier cPlayer.h par la constante MAX_IA_LEVEL). Le temps effectif pris par l'IA est donc fonction de la puissance de calcul de la machine.

Références AlphaBeta:

- http://pageperso.lif.univ-mrs.fr/~liva.ralaivola/lib/exe/fetch.php?id=teaching%3A20122013%3Aprojetalgo&cache=cache&media=teaching:20122013:minmax.pdf
- https://fr.wikipedia.org/wiki/%C3%89lagage_alpha-b%C3%AAta

cOthelloBoard:

Cette classe implémente toutes les primitives nécessaires à la manipulation (*move*(), *set*() , *get*(), ...) et l'affichage du tableau de jeu (*display*()), la vérification des règles de mouvements (*isMovePossible*(), *isMoveValid*(), ...) ainsi que le calcul de la métrique utilisée par l'IA (*getMetric*()).

Interface Graphique:

J'ai choisi l'environnement graphique GLUT (interface openGL) comme dans le projet xxx.

GLUT est une librairie écrite en C et donc présente quelque difficulté pour être encapsulée dans une classe, c'est ce qui m'a décidé a coder l'interface graphique comme une série de fonctions 'encapsulées' dans un fichier plutôt qu'une vraie classe C++, ce qui aurait été plus élégant (on pourrait alors avoir un objet console et un objet graphique avec une interface commune et on choisirait l'une ou l'autre).

Une fois initialisée et la fonction *glutMainLoop*() appelée, GLUT interagit avec le jeu à travers trois fonctions de rappel ('callback') définie dans la fonction *graphicInit*():

- **mouseButtonCallback**: appelée pour chaque évènement de la souris. Elle communique à l'objet *game* la rangée et la colonne correspondant à un lâché du bouton gauche de la souris. Game fournit alors ces informations à l'objet joueur courant pour lui permettre de jouer un coup.

- **displayCallback**: appelée lorsque l'environnement graphique a besoin de redessiner le contenu de la fenêtre et qu'il ne sait pas le faire de manière autonome, ex. initialisation, fenêtre redimensionnée, ... Cette fonction appelle l'objet *game* (qui appelle elle-même l'objet *board*) pour connaitre et afficher le contenu de chaque case ainsi que les messages d'information.

- **timerCallback**: appelée pour forcer une mise à jour de la fenêtre graphique via un appel à *displayCallback*. La fonction se rappelle elle-même toute les 100ms et est appelée pour la première fois par la fonction *graphicInit*().

L'implémentation graphique est rudimentaire; 2D, pas de texture/lumière/..., texte bitmap, pas de menu, ... plus de temps serait nécessaire pour implémenter une interface plus avancées.

Installation et compilation:

Le package g++ doit évidemment être installée pour pouvoir compiler l'application ainsi que glut ; exemple d'installation sur les machines virtuelles Linux Mint :

```
sudo apt-get install g++

sudo apt-get install freeglut3-dev
```

Compilation

Compilation de l'application **projet** (mode console)

```
make projet (ou simplement make)
```

Compilation de l'application **projetgraph** (mode graphique)

```
make projetgraph
```

Nettoyage des fichiers (autres que les fichiers sources)

```
make clean
```

Creation du fichier projet.zip

```
make projet.zip
```

!!! IMPORTANT !!! : Pour passer de l'application « **projet** » à l'application « **projetgraph** », il faut d'abord faire un 'make clean' afin de re-compiler tous les fichiers (sans le flag de compilation GRAPH).

Le fichier Makefile permets également de compiler le programmer avec différentes options, exemples :

```
make MINMAX=1 PROFILE=1 MAX_IA_LEVEL=6

make DEBUG=1
```