Rapport Othello Final

Guiol, Sébastien Mathieu

April 28, 2015

Table des matières

Première partie

Introduction

Chapitre 1

Présentation du Projet

Le projet Othello est un projet algorithme de programmation en langage C requis par la deuxième année de licence informatique à Aix-Marseille Université.
Le but final du projet, qui s'est déroulé en 3 jalons, était d'établir un programme permettant de jouer au jeu d'*Othello*, en developpant une intelligence artificielle suffisamment évoluée pour rivaliser contre un joueur de niveau soutenu.

1.1 Jalon 1

La partie algorithmique du premier jalon a été réduite au minimum, puisque cette premiere phase consistait en la mise en place d'une interface de jeu utilisateur/utilisateur, en implémentant les règles de bases du jeu d'*Othello*.
Cette interface devait être dotée des fonctionnalités suivantes :

- Afficher l'Othellier et le joueur qui a le trait
- Afficher les coups légaux pour le joueur qui a le trait
- Afficher le nombre de points de chaque joueur
- Jouer un coup
- Repérer si un coup est illégal et en informer les joueurs
- Annuler un coup
- Désannuler un coup
- Détecter la fin d'une partie et désigner le vainqueur
- Proposer une nouvelle partie sans avoir à relancer le programme
- Charger une partie à partir d'un fichier (texte)
- Sauvegarder une partie dans un fichier (texte)

1.2 Jalon 2

Dans cette deuxième partie, la partie algorithmique a été plus poussée, puisqu'il s'agissait de mettre en place un jeu utilisateur/ordinateur à l'aide de l'algorithme Min-Max, qui représente les possibilités de jeu comme un arbre et choisit, grâce à une fonction d'évaluation, le meilleur coup à jouer.
A la fin de cette période, l'algorithme Min-Max devait être fonctionnel, posséder une fonction d'évaluation sommaire, et devait pouvoir jouer et surpasser un joueur débutant.

1.3 Jalon 3

Bien que la partie algorithmique, qui est le but premier de cette UE, ait été plus poussée dans la deuxième partie que dans première, cette troisème et dernière phase a été pour moi la plus intéréssante. En effet, ce dernier chapitre avait pour but de mettre en place l'idée de coupures alpha-beta dans l'algorithme Min-Max et, au choix, soit de finir l'implémentation des deux précédentes parties de façon à consolider le programme et à le rendre moins vulnérable aux erreurs, soit d'implémenter un algorithme d'apprentissage génétique pour renforcer la difficulté de jeu en calculant de meilleurs coefficients pour la fonction d'évaluation, de façon à rendre le programme meilleur contre un joueur plus expérimenté.
Pour ma part, j'ai choisi la seconde possibilité, qui était plus adaptée à mon futur projet professionnel, et la réalisation de ce dernier algorithme m'a permis de me rendre compte de l'importance des choix techniques dans l'algorithmique (vitesse d'exécution, occupation mémoire, etc).

Chapitre 2

Le Jeu d'Othello

2.1 Historique

Othello est un jeu de plateau basé sur le modèle du jeu *Reversi*, ce dernier ayant probablement été inventé durant les années 1880 (la date exacte de sa création reste à définr). A cette époque, deux hommes de nationnalité anglaise se disputent la paternité de ce jeu, devenu très populaire en Angleterre. Bien que quelques fabricants de jeux connus, tels que Ravensburger, décident de produire Reversi, celui-ci perd de sa popularité au début du XXème siècle.
C'est au Japon, en 1971, que le Reversi renaît, sous le nom d'*Othello*, réinventé par M. Goro Hasegawa. Quelques règles diffèrent du *Reversi* : seule la position en carré croisé au centre du damier est autorisée comme position de départ, et il est possible d'emprunter des pions à son adversaire si celui-ci est obligé de passer son tour (n'oublions pas que l'*Othello* est un jeu de plateau et possède un nombre de pions fixe au départ : 32 pour chacun des deux joueurs).
En 1976, l'*Othello* se répand dans le monde entier, et le premier championnat du monde d'*Othello* est organisé en 1977.
C'est dans les années 90 que la dimension informatique du jeu d'*Othello* voit le jour, marquant le développement d'algorithmes de plus en plus performants.

2.2 Règles du Jeu

Othello est un jeu de plateau et de stratégie se jouant sur un damier unicolore de 64 cases (8 x 8), que nous appellerons othellier. Pour marquer la possession d'une case, on utilise des pions bicolores : si la face noire est visible, la case appartient au joueur noir, si la face blanche est visible, la case appartient au joueur blanc.

2.2.1 But du Jeu

Le but du jeu est d'avoir sur l'othellier, à la fin de la partie, plus de pions de sa couleur que son adversaire. La partie se termine lorsqu'aucun des deux joueurs ne peut poser de pions (plus aucun coup légal), ou bien lorsque l'othellier est complètement rempli.

2.2.2 Position de Départ

L'othellier est divisé en lignes (de 1 à 8) et en colonnes (de A à H). Pour représenter une case, on utilisera la notation colone-ligne. Par exemple, A1 représente le coin suppérieur gauche et H8 le coin inférieur droit. Au commencement de la partie, chaque joueur possède 2 pions de sa couleur. Les joueurs noir voit ses deux pions pacés en E4 et D5 et les pions du joueur blanc sont en D4 et E5 (figure 2.1).

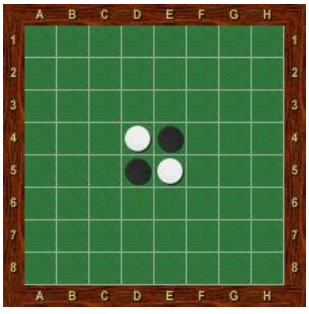


FIGURE 2.1 – Position de Départ

2.2.3 Jouer un Coup

L'interet du jeu d'*Othello* réside essentiellement dans les conditions de placement d'un pion.

En effet, pour placer un pion, un joueur doit vérifier les critères suivants :
— Jouer sur une case vide de l'othellier
— Jouer sur une case adjacente à un pion de couleur adverse

— Encadrer entre le pion qu'il pose et un autre pion de sa couleur un ou plusieurs pions adverses, en formant au moins un alignement continu (pas de case vide)

Si ces critères sont vérifiés, tous les pions de l'adversaire ayant été encadrés sont retournés et prennent la couleur des pions de ce joueur.

2.2.4 Fin de Partie

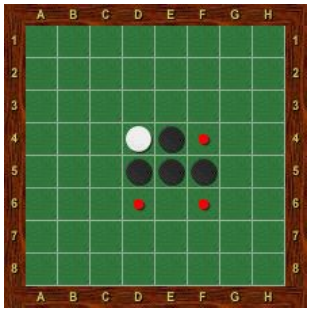


FIGURE 2.2 – Des coups légaux

La partie se finit lorsqu'aucun des deux joueurs ne peut jouer, c'est à dire lorsqu'il n'existe plus aucun coup légal pour chacun. Cela arrive en général lorsque les 64 cases de l'othellier sont remplies, mais il est aussi possible que personne ne puisse jouer avant l'épuisement des cases. En effet, il arrive qu'un joueur bloque le jeu en posant un pion, de telle sorte qu'il n'y ait plus aucun pion de la couleur adverse, par exemple.

Chapitre 3

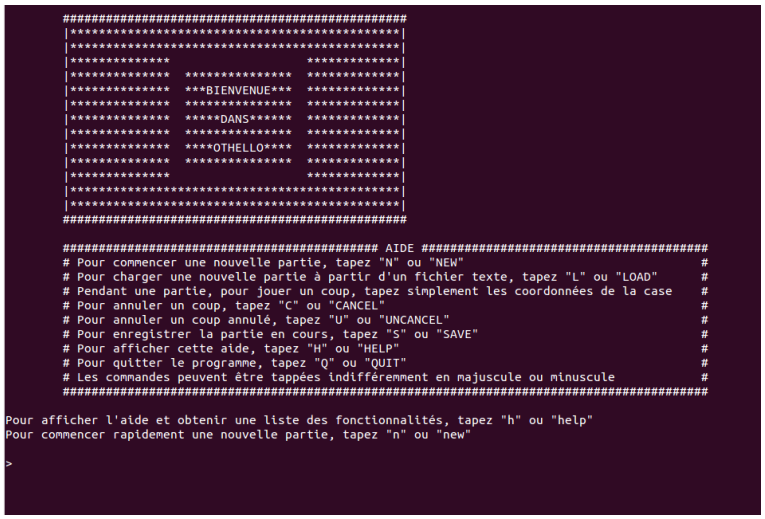
Mode d'Emploi

3.1 Mode Jeu

3.1.1 Compilation et Exécution

Le mode de jeu s'exécute lorsque la macro « GEN_ALGO » est définie à 0.
Pour compiler sous linux, il suffit de taper la commande « make othello » ou bien tout simplement « make ». Le programme se compile alors, et un exécutable « othello » est créé. Pour l'exécuter, il faut donc taper « ./othello ».
La page d'accueil du programme s'affiche alors dans le terminal.

3.1.2 Menu



Le menu d'accueil du programme est composé simplement du titre et d'une aide, pour expliquer à l'utilisateur les commandes existantes et lui permettre de commencer sa partie.
L'entrée des commandes peut se faire avec ou sans majuscule, le programme ne prend pas en compte la casse pour plus de facilité pour les utilisateurs.

3.1.3 Commande N ou NEW

Un appui sur la touche « N »permet de commencer une nouvelle partie. Le programme propose alors 3 modes de jeu :

Humain vs. Humain

Ce premier mode permet à deux utilisateurs de jouer à l'*Othello* l'un contre l'autre. En sélectionnant ce mode par un appui sur la touche « 1 » le programme demande respectivement les noms du joueur noir et du joueur blanc. Pour chacun d'entre eux, il demande confirmation, et si le nom entré n'est pas valide, demande à nouveau à l'utilisateur de l'entrer.
Une fois les deux noms choisis, la partie commence et le programme affiche l'othellier, les carrés, triangles et cercles représentants respectivement les pions noirs, blancs, et les coups légaux pour le joueur qui a le trait.
Sous l'othellier est affiché le score en fonction de la couleur du joueur, et les noms des joueurs relatifs à leur

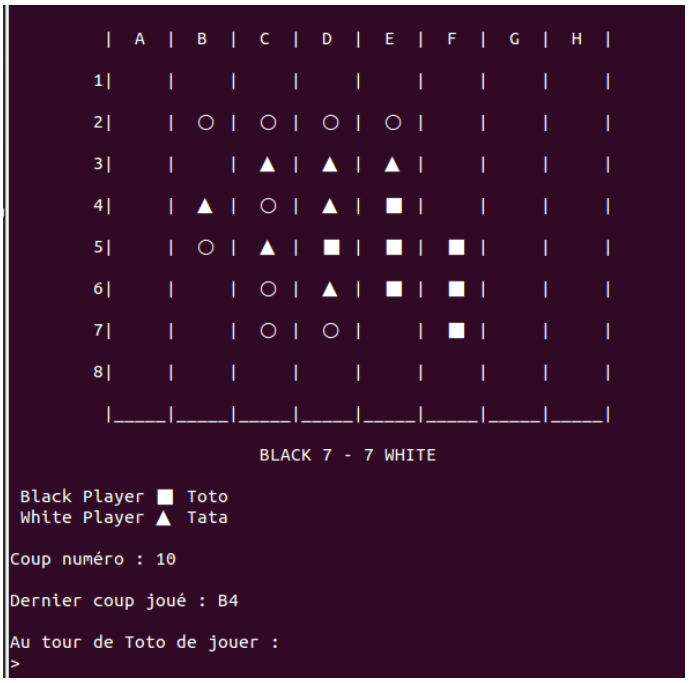
couleur sont rappelés plus bas.
Un curseur indique le numéro du coup joué, le dernier coup joué, et quel est le joueur qui a le trait (« Au tour de [joueur] de jouer »).

Humain vs. Ordinateur

Ce second mode de jeu permet de jouer en mode solo contre l'ordinateur. Ici, le programme demande à l'utilisateur quelle est la couleur de pions que celui-ci souhaite avoir durant la partie, puis demande le nom du joueur pour la couleur selectionnée. Le nom pour l'ordinateur est attribué automatiquement. Tout comme pour le mode précédent, l'othellier et les mêmes informations sont affichés à l'écran.
L'utilisateur va ici affronter l'intelligence artificielle du programme, qui utilise l'algorithme min-max et une fonction d'évaluation pour déterminer le meilleur coup à jouer.
Une fois que l'utilisateur a entré le coup qu'il veut effectuer, l'ordinateur prend immédiatement le relais et place un pion à la case qu'il aura décidé la plus judicieuse. Aucune action n'est requise de la part de l'utilisateur.

Ordinateur vs. Ordinateur

Enfin, ce dernier mode permet uniquement de visionner une partie ordinateur contre ordinateur et sert surtout à des fins de développement. Une fois ce mode choisi, l'ordinateur s'affronte lui même jusqu'à la fin de la partie.



3.1.4 Commande L ou LOAD

Cette commande permet de charger une partie précédemment jouée (finie ou non) à partir d'un fichier texte ayant un format défini.
Le programme demande le nom du fichier texte à charger, et essaie de charger la partie correspondante. Si le fichier n'existe pas, le programme le fait remarquer. Si le fichier existe et représente bien une précédente partie, celle-ci est chargée et l'othellier est affiché. Si le fichier existe bien, mais que son contenu ne représente pas une partie, le comportement est indéfini. Le plupart du temps, le programme considèrera une nouvelle partie, et les premiers mots du programme seront utilisés comme nom pour chacun des deux joueurs.
Le chargement d'une partie à partir d'un fichier texte prend en compte le mode de jeu : si l'un des deux joueurs possède un nom tel que celui défini par l'ordinateur pour le joueur représentant l'ordinateur, alors la partie chargée sera considérée comme un mode Humain contre Ordinateur.
Cette commande est disponible sur le menu d'accueil mais aussi au cours d'une partie. Si appelée durant une partie, la sauvegarde de celle-ci sera demandée.

3.1.5 Commande C ou CANCEL

Durant une partie, pour annuler un coup, il est possible d'utiliser cette commande.
En mode Humain contre Humain, le programme annulera un coup s'il est possible d'en annuler un.

En mode Humain contre Ordinateur, le programme annulera plusieurs coups : celui ou ceux joués par l'ordinateur et celui joué par l'utilisateur, ce façon à ce que le tour lui revienne. Ainsi, il lui sera possible de rejouer à nouveau.
Il est aussi possible d'annuler un coup dans une partie chargée à partir d'un fichier. Dans ce cas, le fichier d'origine ne sera pas modifié mais la partie en mémoire verra un ou plusieurs coups s'annuler, en fonction du mode choisi.
En dehors d'une partie, cette commande ne réalise aucune action.

3.1.6 Commande U ou UNCANCEL

De la même façon que pour la commande « CANCEL », il est possible d'annuler une annulation.
En mode Humain contre Humain, le programme désannulera une seule annulation.
En mode Humain contre Ordinateur, le programme désannulera les actions qui ont été réalisées par l'annulation, c'est à dire qu'il renouvelera la dernière action (humaine) ainsi que tous les coups qu'avait joué l'ordinateur avant l'annulation, jusqu'à ce que le tour revienne au joueur humain.
Il est aussi possible de désannuler un ou plusieurs coup dans une partie chargée, tant qu'il y a eu des coups annulés. La procédure suivie par le programme sera la même que dans une partie normale.
En dehors d'une partie, l'activation de cette commande ne réalise aucune action.

3.1.7 Commande S ou SAVE

Lors du déroulement d'une partie, les utilisateurs peuvent, à leur convenance, choisir d'enregistrer la partie en cours. Pour cela, il suffit de taper la commande « S », ou bien « SAVE »Lors de l'invocation de cette commande, le programme affiche un message de confirmation. Si la réponse est positive, il demande le nom du fichier de sauvegarde.
Si le fichier entré existe déjà, le programme en averti l'utilisateur, puis demande confirmation, puisque la sauvegarde dans un fichier existant supprimera tout type de données présentes dans ce fichier. Si la réponse est positive, la partie est sauvegardée et un message s'affiche dans le terminal.

3.1.8 Commande H ou HELP

Cette commande permet simplement d'afficher une aide sommaire des commandes reconnues par le programme et leur fonction.

3.1.9 Commande Q ou QUIT

Cette commande est disponible aussi bien durant une partie que sur le menu d'accueil du programme. Elle permet de quitter le programme de façon propre, en libérant la mémoire allouée.
Lorsqu'appelée durant une partie, cette commande permet au programme de vérifier que la partie a bien été sauvegardée. Si ce n'est pas le cas, la proposition en est faite.
Une fois les vérifications effectuées, le programme affiche un message et se ferme.

3.1.10 Coordonnées

Pour indiquer au programme, durant une partie, sur quelle case l'utilisateur souhaite poser son pion, il suffit de rentrer les coordonnées de cette case en commençant par la lettre indiquant la colonne, en minuscule ou bien majuscule, puis en indiquant le numéro de la ligne, le tout sans espace.
Si les coordonnées spécifiées correspondent à un coup légal, le coup sera joué, sinon, le programme affichera que le coup n'est pas légal (case déjà occupée ou ne menant pas à un alignement valide).
Si les coordonnées ne correspondent à aucune case ou n'ont pas le format attendu, le programme affichera le message d'erreur suivant « Commande non reconnue ».

3.2 Mode Apprentissage Génétique

Le mode apprentissage génétique s'exécute lorsque la macro « GEN_ALGO » est définie à 1.
La compilation ainsi que l'éxécution se fait de la même façon que le mode de jeu normal.
En mode apprentissage génétique, l'algorithme génétique se lance dès le début de l'éxécution du programme et un message confirmant son éxécution s'affiche. Au fil des générations, les résultats des jeux génomes contre génomes sont affichés, et la fitness est affichée et recalculée après chaque jeu. Selon les valeurs des différentes macros, l'algorithme s'exécute un nombre de générations défini. De même, le nombre d'individus et le nombre de jeu par individu est prédéfini par ces macros.
Se reporter à la section « Apprentissage Génétique » de la partie « Intelligence Artificielle » pour plus d'informations.

Deuxième partie

Architecture du Programme

Chapitre 4

Structures de Données

4.1 Othellier

La structure *Othellier* représente le tableau de jeu de la partie à un instant donné. Elle recense les informations nécéssaires à l'affichage et à l'utilisation du plateau de jeu.

La structure *Othellier* est dotée des éléments suivants :

— *array* : un tableau d'entiers courts à une dimension de taille MAX_CASE (100) pour contenir les valeurs de chaque case. Le tableau représente toutes les cases du damier ainsi que les bords du damier. Les cases 0 à 9, 10, 19, 20, 29, 30, 39, 40, 49, 50, 59, 60, 69, 70, 79, 80 et 89 à 99 représentent donc les bords. Le chiffre des dizaines représente le numéro de la ligne, et le chiffre des unité représente la colonne, avec 1 pour la colonne A jusqu'à 8 pour la colonne H.

— *piecesNb* : un tableau d'entiers courts de taille 2 qui contient, à tout moment, le nombre de pions de chaque joueur, indexé par sa couleur (BLACK = 0, WHITE = 1).

— *turn* : un entier court qui permet de savoir quel est le joueur qui a le trait par rapport à cet othellier. Cela permet, par exemple, de connaitre le trait en fonction d'un othellier préalablement enregistré (pour améliorer la vitesse d'éxécution de la sauvegarde). Cet attribut n'était pas présent dans la première implémentation de la fonction d'annulation (qui recalculait l'état de l'othellier en reparcourant tous les coups existants à chaque appel de l'annulation). Lors de l'ajout de l'intelligence artificielle donnée par l'algorithme Min-Max, les fonctions d'annulation et de désannulation ont été revues, d'où l'apparition de cet attribut de sauvegarde du tour.

```
/** \brief Othellier (game board) structure*/
typedef struct {
    /** Array for the game board */
    short array[MAX_CASE];
    /** Array for the storage of the number of pawn for each player */
    short piecesNb[PLAYER_NB];
    /** Number of the player who has the turn */
    short turn;
} Othellier;
```

FIGURE 4.1 – Structure Othellier

4.2 Player

Cette structure représente les informations utiles à la caractérisation d'un joueur.

La structure *Player* est dotée des éléments suivants :

— *player_name* : une chaine de caractère de taille MAX_NAME (15) qui permet d'enregistrer et de sauvegarder le nom du joueur qu'il définit.

— *player_type* : un entier court qui peut prendre soit la valeur HUMAN (0), soit la valeur COMPUTER(0), et définit si le joueur est un humain ou bien une intelligence artificielle.

```
/** \brief Player structure*/
typedef struct {
    /** String for the storage of the name of one player */
    char player_name[MAX_NAME];
    /** Type of the player (human/computer) */
    short player_type;
    /** Level of the player (beginner/advanced) */
    short player_level;
} Player;
```

FIGURE 4.2 – Structure Player

4.3 Move

Cette structure permet de caractériser un coup : position, couleur, validité.

La structure *Move* est dotée des éléments suivants :

— *position* : un entier court permettant de définir la position d'un coup, c'est à dire à quel endroit dans le tableau de l'othellier le coup va être joué.

— *color* : un entier court désignant la couleur du joueur à qui appartient le coup.

— *flip* : un tableau d'entiers courts définissant s'il existe des retournements possibles de pions concernant le coup, et dans quelles directions.

```
/** \brief Move structure */
typedef struct {
    /** Positionf of the move on the game board (array) */
    short position;
    /** Color of the player who makes the move */
    short color;
    /** Array for the existence of flip if the move is played */
    short flip[MAX_FLIP];
} Move;
```

FIGURE 4.3 – Structure Move

4.4 Game

C'est la structure la plus importante du programme puisque c'est elle qui recense tous les attributs nécéssaires au déroulement d'une partie. De plus, elle caractérise une partie complète et contient toutes les informations pour reconstituer une partie (une sauvegarde, par exemple).

La structure *Game* est dotée des éléments suivants :

— *cursor* : un entier court permettant de compter le numéro du coup courant. Bien que son but originel ait été simplement de compter le nombre de coups, il s'est avéré extrêmement utile comme curseur du tableau de coups pour repérer le coup actuel, et a servi dans bien des fonctionnalités, telles que l'annulation et la désannulation.

— *movesCursor* : un entier court repérant la dernière case du tableau de coups. Il est utile dans un langage procédural tel que le C, pour lequel la taille d'un tableau n'est pas donnée par une fonction (à l'inverse de la programmation orientée objet). Cet attribut permet donc de parcourir le tableau de coups sans dépasser sa fin, et est utile pour toutes les actions qui requièrent le parcours de ce tableau (en particulier annulation et désannulation).

— *boardsCursor* : de même que *movesCursor*, cet attribut est un entier court repérant la fin du tableau des othelliers. Son utilité est la même que son homologue.

— *turn* : un entier court désignant la personne qui a le trait au moment courant de la partie, en opposition à l'attribut *turn* de la structure *player*.

— *saved* : un entier court qui désigne le statut de la partie en cours : sauvegardée, ou non.

— *p1* : une structure *Player* désignant, ici, le joueur 1 et tous ses attributs. Par convention, le joueur 1 représente la couleur noire.

— *p2* : de même, une structure *Player* désignant le joueur 2 et ses attributs. Par convention, le joueur 2 représente la couleur blanche.

— *moves* : un tableau de structures *Move*, indexé entre 0 et *movesCursor*, qui permet de sauvegarder en mémoire tous les coups joués depuis le début de la partie.

— *oths* : de même que précédemment, un tableau d'*Othellier* permettant de sauvegarder tous les états de l'othellier à chaque coup effectué.

— *oth* : une structure *Othellier* désignant ici l'othellier courant et son état.

— *coefs* : enfin, un tableau de flottants de taille COEF_NB (4) recensant les coefficients à utiliser dans la fonction d'évaluation. Ceux-ci sont définis par apprentissage génétique, ou bien, si celui-ci n'a pas été éffectué, ils sont initialisés à 1 par défaut.

```
/** \brief Game structure */
typedef struct {
    /** Number of the move */
    short cursor;
    /** Cursor for the last cell of the moves array */
    short movesCursor;
    /** Cursor for the last cell of the boards array */
    short boardsCursor;
    /** Number of the player who has to play */
    short turn;
    /** Is the game saved ? 1 for yes, 0 for no */
    short saved;
    /** Player 1 */
    Player p1;
    /** Player 2 */
    Player p2;
    /** Moves array (for saving) */
    Move** moves;
    /** Game boards array (for saving) */
    Othellier** oths;
    /** Othellier (game board) */
    Othellier* oth;
    /** Coefficients used by the evaluation function */
    float coefs[COEF_NB];
} Game;
```

FIGURE 4.4 – Structure Game

4.5 Genome

Cette structure est utile dans le cas d'un apprentissage génétique pour simuler une « sélection natuelle » des coefficients de la fonction d'évaluation.

La structure *Genome* est dotée des éléments suivants :

— *coefs* : un tableau de flottants de taille COEF_NB (4) recensant les coefficients à utiliser dans la fonction d'évaluation, de façon a être évalués par l'algorithme génétique.

— *fitness* : un entier court représentant le degré d'adaptabilité (fitness) de l'individu en question dans une génération. Plus cet entier est élevé, plus l'individu est adapté à sa génération et plus les coefficients sont adaptés à l'exécution de l'intelligence artificielle du programme.

```
/** \brief Genome structure */
typedef struct {
    /** Coefficients array */
    float coefs[COEF_NB];
    /** Fitness of the genome */
    short fitness;
} Genome;
```

FIGURE 4.5 – Structure Genome

4.6 Population

Cette structure simule une population de coefficients, donc une génération complète. Elle n'est pas indispensable mais permet une division des génomes par génération.

La structure *Population* est dotée des éléments suivants :

— *genomes* : un tableau de génome recensant tous les individus (génomes) d'une génération.

— *genNb* : un entier court déterminant le nombre d'individus dans la population.

```
/** \brief Population structure */
typedef struct {
    /** Genomes array */
    Genome* genomes;
    /** Number of genomes in the population */
    short genNb;
} Population;
```

FIGURE 4.6 – Structure Population

4.7 Node

Une structure *Node* avait été mise en place, ainsi que toutes les fonctions relatives à son fonctionnement, mais celle-ci n'a jamais été utilisée, par manque de temps, mais aussi après réflection. Elle avait été pensée de façon à créer et sauvegarder l'arbre généré par Min-Max de façon à gagner en rapidité d'exécution, et ajouter à chaque appel de Min-Max 1 étage de plus que précédemment.

Cependant, cela n'aurait pu être possible au vu de l'espace mémoire que ce fonctionnement requiert. En effet, pour construire un arbre de taille 6, il aurait fallu en moyenne 62 (taille en octets d'un élément de la structure *Node*) * 8^6 (en considérant qu'il y a en moyenne 8 coups possibles) soit 16252928 octets, soit près de 16 Mo. En considérant que la taille de l'arbre augmente de 1 à chaque coup joué, la profondeur maximale de l'arbre aurait été de 33, ce qui représente une espace mémoire inconsidérable pour des ordinateurs de bureau.

L'idée originale a donc été abandonnée, mais il aurait été intéressant d'essayer de l'adapter de façon à avoir des résultats viables pour un ordinateur personnel, et ainsi pouvoir comparer les différentes vitesses d'exécution pour chacune des implémentations.

Chapitre 5

Implémentation

5.1 Ordre

Pour chacune des trois phases de la réalisation de ce projet, l'ordre d'implémentation à été différent.

5.1.1 Jalon 1

Dans cette première partie, j'ai commencé par réfléchir aux structures de données que j'allais utiliser pour concervoir tout le programme. Mis à part quelques changements mineurs, elles n'ont pas trop évoluées, ce qui est une bonne chose. Ensuite, j'ai réfléchi sur papier aux principaux algorithmes de jeu, comme la détection des coups légaux, le retournement des pions et le traitement d'un coup. Puis j'ai commencé par implémenter les deux premiers, avant même de créer une interface d'utilisation et d'afficher l'othellier. Une fois ces algorithmes testés et validés, j'ai implémenté l'affichage de la grille et des pions, puis le traitement d'un coup, ce qui a été rapide puisque réfléchi précédemment. J'ai donc pu tester ce dernier grâce à l'affichage, et ai pu améliorer les intéractions avec l'utilisateur. Enfin, j'ai ajouté les fonctionnalités d'annulation et de désannulation, de sauvegarde et de chargement, en prenant bien soin de tester tous les cas possibles pour ces deux dernières. C'est pourquoi, pour le chargement, j'ai préféré ne pas utiliser la fonction *scanf* du langage C, qui a souvent un comportement indéfini.

5.1.2 Jalon 2

Avant de commencer ce deuxième jalon, je me suis rendu compte qu'il était possible d'optimiser ma détection de coups légaux, c'est pourquoi celle-ci a été revue. J'ai aussi remarqué que ma fonction *main* était un peu chargée, c'est pourquoi j'ai opté pour une vision interpréteur, avec des évènements reliés aux différentes commandes disponibles dans le jeu. Aujourd'hui, la fonction principale ne contient plus que la reconnaissance des évènements.
Ensuite, j'ai réfléchi à une implémentation possible de l'algorithme Min-Max, en utilisant la structure *Node* évoquée dans le chapitre précédant. Après quelques calculs, je me suis rendu que cela aurait été plus compliqué et qu'il était préférable de garder cette idée pour la suite. J'ai donc retenu une interprétation plus simple, en gardant à l'idée qu'elle pourrait facilement être adaptée à la génération d'un arbre.
Enfin, j'ai implémenté l'algorithme, élaboré une fonction d'évaluation prenant en compte la force des positions, la mobilité et le nombre de pions, et, étant un peu en avance, j'ai ajouté la simplification par les coupures Alpha-Beta, et ai modifié les fonctions d'annulation et de désannulation.

5.1.3 Jalon 3

Pour cette dernière partie, j'avais beaucoup d'améliorations à effectuer. J'ai du me concentrer sur les principales. J'ai donc commencé par ajouter un algorithme d'apprentissage génétique, que j'ai d'abord testé sur quelques générations. J'ai ensuite essayé de réduire le nombre d'allocations dynamiques, mais, l'ayant pourtant divisé par 4, n'ai pas constaté de réduction notable du temps d'exécution (testé sur plusieurs parties Ordinateur contre Ordinateur).
Enfin, j'ai modifié ma fonction d'évaluation de façon à ce qu'elle prenne en compte un pattern particulier au jeu d'Othello. Puis, après quelques calculs (détaillés dans le chapitre « Apprentissage Génétique »), j'ai exécuté l'algorithme génétique.

5.2 Changements

Les principaux changements effectués tout au long de ce projet sont recensés ci-dessous :

— *Détection des coups légaux* : auparavant, la détection des coups légaux se faisait de façon récursive. Or, il était tout à fait possible de réaliser cela de façon itérative. De plus, une fonction récursive utilise beaucoup plus d'espace mémoire (utilisation intense de la pile) et met donc plus de temps à s'exécuter. C'est pourquoi, les 8 fonctions (une fonction par direction) de détection des coups légaux ont été réduites à une seule prenant la direction en paramètre, et leur caractère récursif s'est vu transformé en itératif.

— *Annulation* : avant modification, l'annulation d'un coup se faisait en supprimant le dernier coup du tableau de coups, en réinitialisant l'othellier à sa configuration de départ, en en reparcourant chacun des coups de façon à recalculer l'état précédent de l'othellier. Cette méthode, bien que fonctionnelle, était gourmande en ressources et plus spécifiquement en temps. C'est pourquoi j'ai décidé d'enregistrer à chaque coup l'othellier courant, ce qui a permis de rétablir l'othellier rapidement lors d'une annulation. Cela a été extrèmement bénéfique pour l'algorithme Min-Max, qui a vu son temps d'exécution divisé par plus de 2.

— *Désannulation* : à l'instar de l'annulation, la désannulation a aussi du être modifiée pour fonctionner avec ce nouveau principe.

— *Espace mémoire* : pour diminuer l'espace mémoire occupé par le programme, tous les attributs qui le permettaient ont été transformés en entiers courts au lieu d'entiers normaux, ce qui permet un gain de place de 2 octets par entier sur la plupart des systèmes. L'utilisation mémoire a donc été divisée par 2, ce qui a largement rattrapé l'espace perdu par la modification de l'annulation (206 octets par othellier * 64 othelliers stockés au maximum = un peu moins de 13 Ko pour stocker l'intégralité des othelliers par partie).

— *Fonction d'évaluation et Min-Max* : pour pouvoir utiliser des coefficients particuliers pour l'algorithme génétique, j'ai du modifier ma fonction d'évaluation et l'algorithme Min-Max de façon à ce qu'ils prennent en paramètre un tableau de coefficients. Ainsi, il a été possible de faire jouer certains coefficients contre d'autres.

Chapitre 6

Pseudo-Code des Algorithmes

6.1 Traitement d'un Coup

```
move_processing()
Entree : Game, Coordinates
Si les coordonnees n'ont pas un format correct
      retourner
var position = Convertir les coordonnees
var move = Creer un coup (position)
Si move n'est pas legal
      supprimer move
      retourner
Sinon, jouer le coup move
      Enregistrer move dans Game
      Mettre a jour l'Othellier dans Game
      retourner
```

6.2 Jouer un Coup

```
play_move()
Entree : Game, Move
Sortie : nombre de pions retournes
var compteur = 0
Augmenter le nombre de pions du joueur associe a move
compteur = compteur + 1
Pour chaque direction a partir de la position de move, faire:
      Tant que la case suivante dans la meme direction est de couleur opposee a celle de
move
        Changer la couleur de la case
        Incrementer le nombre de pions du joueur du coup
        Decrementer le nombre de pions de l'adversaire
        compteur = compteur + 1
      Fin tant que
Fin pour
retourner compteur
```

6.3 Coup Légal

```
is_legal()
Entree : Othellier, Move
Sortie : 1 si le coup est legal, 0 sinon
Si la case a la position associee a move est deja occupee
retourner 0
Mettre a jour les retournements de move
Pour chaque case du tableau de retournements de move, faire :
```

```
            Si la case vaut 1   // S'il y a retournement
              retourner 1
      Fin pour
      retourner 0
```

6.4 Mise à Jour des Retournements

```
      update_flip()
      Entree : Othellier, Move
      Sortie : tableau des retournements mis a jour
      Si la case a la position associee a move est un bord
            retourner
      Pour chaque direction
            Si la case suivante dans la direction voulue est de couleur opposee a celle de
    move
              Verifier qu'il y a un retournement, mettre la case a 1 si c'est le cas
            Sinon, mettre la case de la direction du tableau de retournements de move a 0
       retourner

```

6.5 Vérification des Retournements

```
      has_alignment()
      Entree : Othellier, position, color, direction
      Sortie : 1 s'il y un retournement, 0 sinon
      var shift = position
      Tant que la case shift de l'othellier est de la couleur opposee a color
            Si la case suivante (shift+direction) de l'othellier est de la couleur color // il
     y a eu retournement
              retourner 1
            shift = shift + direction
      Fin tant que
      retourner 0
```

6.6 Annulation Coup

```
      cancel_last_move()
      Entree : Game
      Sortie : Game avec un coup annule, 1 si un coup a ete annule, 0 sinon
      S'il n'y a pas de coup a annuler
            retourner 0
      Decrementer le curseur de jeu
      Copier le materiel du coup precedent de chaque joueur dans l'othellier courant
      Copier la valeur de chaque case du coup precedent dans l'othellier courant
      Copier le joueur qui avait le trait le tour precedent dans le jeu courant
      retourner 1
```

6.7 Désannulation Coup

```
      uncancel_move()
      Entree : Game
      Sortie : Game avec un coup desannule, 1 si un coup a ete desannule, 0 sinon
      S'il n'y a pas de coup a desannuler
```

```
        retourner 0
    Incrementer le curseur de jeu
    Copier le materiel du coup suivant de chaque joueur dans l'othellier courant
    Copier la valeur de chaque case du coup suivant dans l'othellier courant
    Copier le joueur qui a le trait le tour suivant dans le jeu courant
    retourner 1
```

6.8 Min-Max

```
    Entree : Maximize, Game, playerAi, depth, height, alpha, beta, coefs
    Sortie : Le meilleur score ou le meilleur coup a jouer si l'on est en haut de l'arbre
    Si end_of_game ou depth < 1
        retourner evaluation();
    var pos tabLegal = tableau des coups legaux
    var tabLength = taille du tableau des coups legaux
    var evaluation, evalMax, position
    Si Maximize // noeud IA
        eval = -infini
        Pour chaque case de tabLegal // pour chaque coup legal
          Jouer le coup
          Si le tour est passe a l'adversaire
            evaluation = min-max(NO, game, playerAi, depth-1, height, alpha, beta, coefs)
          Sinon // Si un joueur doit passer son tour
            evaluation = min-max(YES, game, playerAi, depth-1, height, alpha, beta, coefs)
          Annuler le coup
          Si evaluation > evalMax
            evalMax = evaluation
            position = coup joue precedemment
          Si evalMax >= alpha
            alpha = evalMax
          Si beta <= alpha // Coupure beta
            break
        Fin pour

    Si Maximize // noeud IA
        eval = +infini
        Pour chaque case de tabLegal // pour chaque coup legal
          Jouer le coup
          Si le tour est passe a l'adversaire
            evaluation = min-max(YES, game, playerAi, depth-1, height, alpha, beta, coefs)
          Sinon // Si un joueur doit passer son tour
            evaluation = min-max(NO, game, playerAi, depth-1, height, alpha, beta, coefs)
          Annuler le coup
          Si evaluation < evalMax
            evalMax = evaluation
            position = coup jou  precedemment
          Si evalMax <= beta
            beta = evalMax
          Si beta <= alpha // Coupure alpha
            break
        Fin pour

    Si depth = height // Si l'on est en haut de l'arbre
        retourner position
    Sinon // Min-max n'est pas termin
        Retourner evalMax

```

6.9 Fonction d'Evaluation

```
    evaluation_function()
    Entree : Game, color, coefs[]
    Sortie : Evaluation heuristique de l'etat de jeu
    var evaluation
    evaluation = evaluation + coefs[0] * (nombre de pions de color - nombre de pions de l'
  adversaire)
```

```
    evaluation = evaluation + coefs[1] * mobilite de color
    evaluation = evaluation - coefs[1] * mobilite de l'adversaire
    evaluation = evaluation + coefs[2] * force des positions de color
    evaluation = evaluation - coefs[2] * force des positions de l'adversaire
    evaluation = evaluation + coefs[3] * evaluation du pattern des coins de forme carree de color
    evaluation = evaluation - coefs[3] * evaluation du pattern des coins de forme carree de l'adversaire
    retourner evaluation

```

6.10 Algorithme Génétique

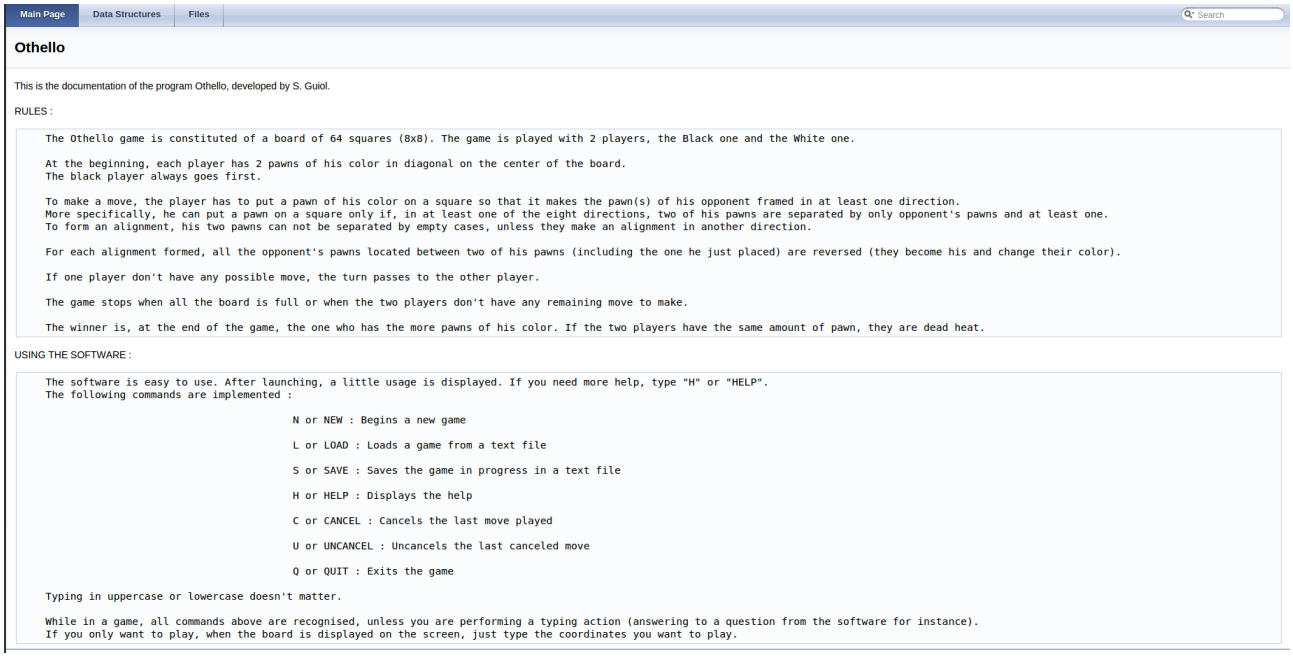
```
    genetic_algorithm()
    Entree :
    Sortie : meilleurs coefficients
    var population = creer une population aleatoirement
    var population_suivante
    Pour chaque generation (n)
          Faire jouer chaque individu de population contre k autres (k < n)
          Calculer la fitness de chacun
          population_suivante = reproduction de population
          Pour chaque individu i
          i = mutation
          Fin pour
          Supprimer population
          population = population_suivante
     Fin pour
     var bestCoef = population[0]
     Pour chaque individu i de population
          Si fitness(population[i]) > fitness(bestCoef)
          bestCoef = population[i]
    retourner bestCoef
```

Chapitre 7

Etat des Fonctionnalités

7.1 Description des Fonctions

Une documentation exhaustive des fonctions est disponible dans la documentation du programme en cliquant ICI.



Main Page | Data Structures | Files | Search

Othello

This is the documentation of the program Othello, developed by S. Guiol.

RULES :

```
The Othello game is constituted of a board of 64 squares (8x8). The game is played with 2 players, the Black one and the White one.

At the beginning, each player has 2 pawns of his color in diagonal on the center of the board.
The black player always goes first.

To make a move, the player has to put a pawn of his color on a square so that it makes the pawn(s) of his opponent framed in at least one direction.
More specifically, he can put a pawn on a square only if, in at least one of the eight directions, two of his pawns are separated by only opponent's pawns and at least one.
To form an alignment, his two pawns can not be separated by empty cases, unless they make an alignment in another direction.

For each alignment formed, all the opponent's pawns located between two of his pawns (including the one he just placed) are reversed (they become his and change their color).

If one player don't have any possible move, the turn passes to the other player.

The game stops when all the board is full or when the two players don't have any remaining move to make.

The winner is, at the end of the game, the one who has the more pawns of his color. If the two players have the same amount of pawn, they are dead heat.
```

USING THE SOFTWARE :

```
The software is easy to use. After launching, a little usage is displayed. If you need more help, type "H" or "HELP".
The following commands are implemented :

                                        N or NEW : Begins a new game

                                        L or LOAD : Loads a game from a text file

                                        S or SAVE : Saves the game in progress in a text file

                                        H or HELP : Displays the help

                                        C or CANCEL : Cancels the last move played

                                        U or UNCANCEL : Uncancels the last canceled move

                                        Q or QUIT : Exits the game

Typing in uppercase or lowercase doesn't matter.

While in a game, all commands above are recognised, unless you are performing a typing action (answering to a question from the software for instance).
If you only want to play, when the board is displayed on the screen, just type the coordinates you want to play.
```

7.2 Tableau des Fonctionnalités

7.2.1 Jeu

N	Nom Fonctionnalité	Commentaires
1.1	Affichage du tableau de jeu, scores et joueur qui a le trait	Affiche l'othellier, le joueur qui a le trait, les scores, une légende concernant les pions, le numéro du coup et le dernier coup joué
1.2	Initialisation du jeu et de l'othellier	Initialise l'othellier selon les règles de l'Othello, permet de choisir entre 3 modes de jeu
1.3	Détection du joueur qui a le trait	Mise à jour automatique du joueur qui a le trait et affichage de celui-ci
2.1	Sauvegarde	Sauvegarde une partie à la demande de l'utilisateur (propose si la fin de partie est atteinte), teste si le fichier texte n'existe pas déjà, demande confirmation à l'utilisateur
2.2	Chargement	Charge une partie à partir d'un fichier, vérifie si le fichier existe, si l'accès en lecture est donné. Si le fichier ne contient pas de partie sauvegardée, charge une nouvelle partie la plupart du temps, sinon comportement non défini
3.1	Annulation	Annule le nombre de coups nécéssaire pour que le tour revienne au joueur si jeu contre ordinateur, annule 1 coup si jeu contre humain
3.2	Désannulation	Désannule le nombre de coups nécéssaire pour que le tour revienne au joueur si jeu contre ordinateur, annule 1 coup si jeu contre humain
4.1	Vérification des coordonnées	Vérifie que les coordonnées entrées ont le bon format et appartiennent bien à la grille, vérifie que les coordonnées représentent un coup légal
4.2	Conversion des coordonnées	Convertit les coordonnées après avoir vérifié qu'elles ont le bon format
5	Détection d'un coup légal	Détecte tous les coups légaux pour le joueur qui a le trait
6	Traitement d'un coup	Vérifie les coordonnées et la légalité du coup avant d'effectuer l'action. Avertit en cas de coup non légal ou mauvais format
7.1	Détection de fin de partie	Détection automatique de fin de partie
7.2	Actions à réaliser lors de la fin du jeu	Demande pour sauvegarde et enregistrement de la partie, puis propose de recommencer une nouvelle partie, permet de changer les couleurs et les noms des joueurs
7.3	Quitter le programme	Vérifie si une partie est en cours et propose sa sauvegarde si cela n'a pas déjà été fait. Désalloue la mémoire avant de quitter
8.1	Intéraction avec l'utilisateur	Ne prend pas en compte la casse concernant les commandes et les coordonnées (conversion en majuscule automatique)
8.2	Vérification de la validité d'un nom	Vérifie que le nom ne contienne que des lettres et des chiffres, sinon repropose
8.3	Vider le Buffer (stdin)	Vide le buffer à chaque fin d'entrée pour éviter le report des caractères non pris en compte dans une future entrée de texte

7.2.2 Intelligence Artificielle

9	Jouer un coup – IA	
9.1	Recherche d'un coup	Tient compte du saut de tour potentiel pour min-max, fonctionne avec le joueur NOIR comme le joueur BLANC
9.2	Evaluation d'un coup	Fonction d'évaluation avec coefficients calculés par apprentissage génétique
9.3	Profondeur de recherche	Plus de gestionnaire de profondeur depuis l'algorithme génétique
9.4	Rapidité de recherche	Amélioration avec les coupures Alpha-Beta, fonction d'annulation modifiée pour accélération
10.1	Annulation IA	Annule de façon à revenir au tour du joueur humain : event_cancel()
10.2	Désannulation IA	Désannule de façon à revenir au tour du joueur humain : event_uncancel()
11	Jeu Ordinateur VS. Ordinateur	Une troisième possibilité de jeu : ordinateur contre ordinateur
12	Algorithme Génétique	Simule une « sélection naturelle » pour trouver les meilleurs coefficients à utiliser dans la fonction d'évaluation

Troisième partie

Intelligence Artificielle

Chapitre 8

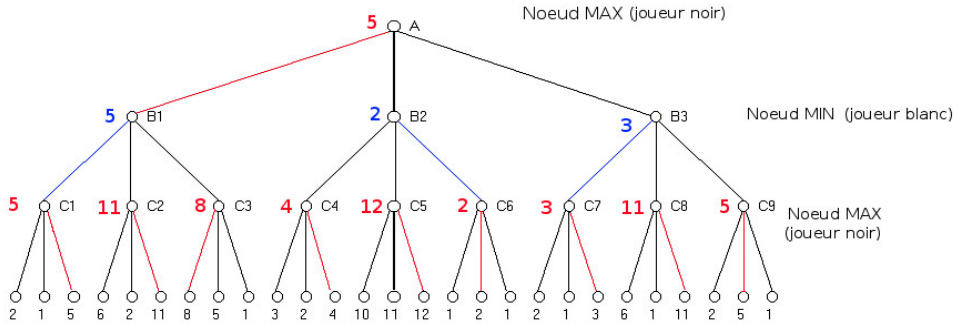
Algorithme Min-Max

8.1 Principe

L'algorithme Min-Max (ou MiniMax en anglais) est un algorithme de sélection de meilleur coup dans les jeux à deux joueurs dits « à somme nulle ». Son principe est simple : il parcourt toutes les possibilités d'action pour chaque joueur, sur un nombre d'étages donné, et choisi la meilleure possibilité en maximisant les gains potentiels du joueur et en minimisant ceux de l'adversaire.
Ainsi, l'algorithme Min-Max permet de choisir non seulement le coup à jouer menant à la victoire, mais aussi le coup à jouer menant au minimum de gains pour l'adversaire. Ainsi, entre une possibilité menant à, par exemple, 4 victoires et une defaîte, et une possibilité menant à 1 victoire et 4 matchs nuls, l'algorithme préfèrera la deuxième, qui ne mènera en aucun cas à la défaite !

8.2 Exemple

Voici une explication sur un exemple :



Ici, c'est au tour du joueur noir de jouer. L'algorithme va chercher le meilleur coup potentiel à effectuer en opérant sur 3 étages.
Premièrement, l'algorithme va simuler un coup en B2, puis continuer pour toutes les possibilités qui s'offrent à lui. Il va donc simuler C1, C2 et C3. En jouant en B2 puis en C1, il y a 3 possibilités : un gain de 2, un gain de 1 ou un gain de 5. Comme nous sommes dans un noeud Max (couleur rouge, c'est au joueur noir de jouer), l'algorithme va maximiser les gains qu'il peut faire. Ainsi, il fera remonter la plus grande valeur, soit 5.

De même, pour C2, la valeur qui remonte sera 11, et 8 pour c3.
Maintenant, on remonte d'un étage. En B1, il y a 3 possibilités : 5, 11 et 8. Ici, c'est au joueur blanc de jouer (l'adversaire), il n'est donc pas possible de choisir à la place du joueur blanc ce qu'il va jouer. Par défaut, le programme choisit donc la valeur la plus basse, qui constitue donc le plus grand avantage pour le joueur blanc (les avantages de l'un sont les désavantages de l'autre). L'algorithme remonte alors la plus petite valeur des trois, soit 5.
Il en est de même pour les autres branches, et, en A, il reste à la fin de l'algorithme 3 valeurs : 5, 2 et 3. Nous sommes dans un noeud max, l'algorithme choisira la plus grande valeur (trait rouge), et jouera donc en B1, le coup qui est considéré comme le meilleur à jouer.

8.2.1 Joueur à Evaluer

Après cette explication, il est clair que la fonction d'évaluation doit obligatoirement évaluer le joueur qui a appelé l'algorithme. En effet, si l'on évalue l'état de jeu pour le joueur qui a le trait, on aura une valeur positive pour l'adversaire au niveau d'un noeud MIN, et l'algorithme choisira la plus petite valeur (minimisation), soit dans ce cas précis, un désavantage pour l'adversaire. Or, on ne peut décider à l'avance de quel coup l'adversaire va jouer, c'est pourquoi on choisit par défaut le meilleur coup pour l'adversaire, en supposant qu'il prenne à chaque fois le meilleur coup. Ainsi, il est primordial d'effectuer la fonction d'évaluation de la même façon, sans tenir compte du joueur qui a le trait, de la profondeur ou du caractère (MAX ou MIN) du noeud. La fonction d'évaluation doit donc, peu importent les circonstances, évaluer de la même façon, c'est à dire le joueur qui a appelé l'algorithme Min-Max en positif, et l'adversaire en négatif.
Cette remarque souligne aussi la nécéssité d'effectuer une différence lors de l'évaluation. En effet, bien qu'il soit profitable de posséder, par exemple, 1 coin, si l'adversaire en possède 2, cette situation perd grandement son intérêt. Il est donc nécéssaire de soustraire les gains de l'adversaire aux gains du joueur de façon à évaluer objectivement l'état de jeu en tirant pleinement parti des avantages des jeux à somme nulle.

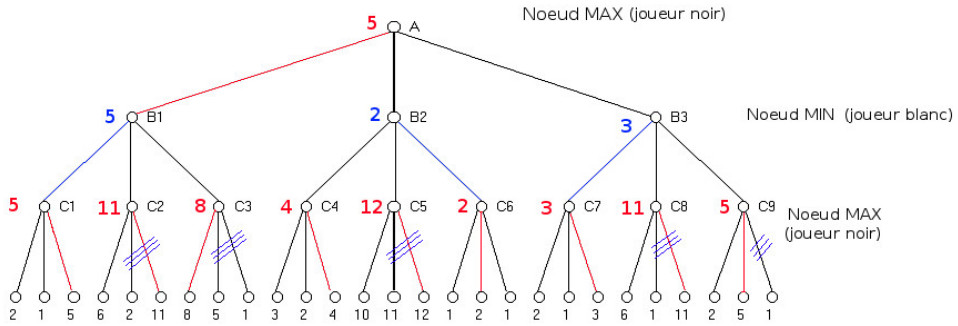
8.3 Elagage Alpha-Beta

8.3.1 Principe

Le principe de l'élagage Alpha-Beta (ou Alpha-Beta Pruning en anglais) est assez simple à comprendre mais requiert un certain temps pour visualiser ce qu'il se passe en temps réel, et pour comprendre comment deux valeurs seulement sont capables de faire réduire le temps d'exécution de Min-Max aussi considérablement.
Le principe repose sur le fait que, lorsqu'on cherche pour un noeud à maximiser une valeur et que l'on trouve, un étage plus bas (donc dans un noeud MIN), une valeur inférieure à celle que l'on souhaite, il n'est plus nécéssaire de continuer d'analyser cette branche. En effet, en minimisant la valeur, on va obtenir cette dernière valeur, plus petite que la précédente, ce qui de toute évidence ne servira pas puisque le noeud précédent demande de maximiser la valeur. On aura donc exploré des branches inutilement.
Alpha-Beta prend en compte ce fait pour réduire le nombre de branches à parcourir. Cela est autant valable pour la minimisation que pour la maximisation.

8.3.2 Exemple

Voici un exemple pour illustrer ce principe :



Ici, l'algorithme parcourt B1, puis C1. Il remonte en B1, puis descends en C2. C2 est un noeud MAX, il va donc prendre la plus grande valeur. Mais, arrivé à la valeur 6, il ne continuera pas à explorer les branches de C2, puisque 6 est supérieur à 5 (la précédente valeur trouvée en C1 est remontée en B1 pour le moment) et que B1 est un noeud MIN. En effet, les seules valeurs utiles en C2 seraient des valeurs plus petites que 5, pui que B1 a pour but de prendre la plus petite de ces valeurs. Il est donc inutile de continuer la recherche à ce niveau, on sait déjà que la valeur qui va remonter en C2 (6 ou plus) ne sera pas retenue en B1 (il existe déjà une valeur inférieure à 6 qui sera sélectionnée à sa place).

Chapitre 9

Apprentissage Génétique

9.1 Principe

Le principe de l'apprentissage génétique est de déterminer, par simulation de « sélection naturelle », certaines valeur de façon à rendre une intelligence artificielle plus forte. Dans le cas de l'othello, cette apprentissage génétique a pour but de déterminer les coefficients à appliquer dans la fonction d'évaluation les plus justes possible, de façon à relever le niveau de Min-Max pour la même profondeur.
En effet, avoir une fonction d'évaluation complète est primordial pour donner une certaine force à l'intelligence artificielle, mais l'application de coefficients à chaque attribut de la fonction d'évaluation l'est encore plus. Il n'est pas possible pour un humain de juger de la justesse des coefficients, et le meilleur moyen de les déterminer est de simuler des parties pour savoir quelle combinaison s'adaptera le mieux à de vraies parties contre des humains.
Pour déterminer ces coefficients, il est donc nécéssaire de démarrer avec un population de coefficients générés aléatoirement (pour le plus d'homogénéité possible), puis d'évoluer petit à petit, en éliminant les individus les moins adaptés aux parties, de façon à converger vers de meilleurs individus et donc de meilleurs coefficients.
Evidemment, plus on a de coefficients à appliquer sur de petits attributs, plus l'évaluation est forte et juste, mais cela augmente considérablement le temps de l'apprentissage.
En effet, si l'on a peu de coefficients, on a une plus petite palette d'individus différents, et il est donc plus facile d'en tester une majorité. A l'inverse, si l'on a une grande palette d'individus, le temps d'exécution se devra d'être beaucoup plus long puisqu'il faudra plus de générations pour arriver au but final.

9.2 Réflexion et Exécution

Dans le but d'obtenir les meilleurs coefficients possible et de façon à pouvoir faire tourner l'algorithme en temps viable, j'ai décidé d'appliquer 4 coefficients à ma fonction d'évaluation. Ces coefficients représentent :

- La mobilité
- La force donnée par les positions des pions en fonction de chaque case
- La différence du nombre de pions
- Un pattern particulier prenant en compte les différentes extensions de carrés de même couleur dans les coins

Les différents coefficients peuvent prendre toutes les valeurs en 0 et 2.5, avec des pas de 0.1, ce qui donne 26 valeurs différentes par coefficients.
Il y a 4 coefficients, ce qui signifie que le nombre maximal d'individus différents est de 26^4 (environ 450,000).
Comme la durée moyenne d'une partie avec ce programme est de 98 secondes (voir rubrique « Statistiques »), j'ai décidé de lancer l'algorithme génétique sur 35 générations, avec 25 individus par génération et 10 jeux par individu, de façon à obtenir une fitness plus précise. Cela représentait un temps total d'éxécution de 9 jours environ.
Malheureusement, à cause d'une panne encore indéfinie, l'ordinateur sur lequel tournait le programme s'est arrété au bout de 15 générations. Les résultats étants trop peu satisfaisant à mon goût, j'ai relancé l'algorithme sans perdre les données déja précalculées sur 20 générations, mais en jouant seulement 3 jeux au lieu de 10 de façon à gagner en rapidité, le temps restant ne me permettant pas de finir avec la configuration d'origine.
Durant ce calcul, j'ai pu remarquer que les coefficients convergeaient petit à petit et se stabilisaient, et que la force de l'intelligence artificielle du programme s'accroissait (capacité à vaincre un joueur confirmé).

9.3 Implémentation

Les points forts de l'algorithme d'apprentissage génétique résident dans 2 particularités de l'implémentation : le nombre d'individus, de générations et de jeux par individus, comme discuté plus haut, et l'implémentation de la sélection.
La sélection des individus pour la génération suivante a été faite sur le principe de la roue de lotterie biaisée. En d'autres termes, on tire aléatoirement les individus à sélectionner, mais on associe une plus grande importance à ceux qui ont la fitness la plus élevée. Ainsi, il y a une plus grande probabilité de tirer les individus les plus adaptés que ceux qui ne le sont pas.
L'implémentation de la reproduction importe peu, tant qu'on réutilise les coefficients des parents pour donner un enfant, les résultats au bout de n générations seront 'équivalents, suivant le principe de la sélection naturelle.
La fitness a été implémentée de façon à laisser la possibilité à des individus faibles de passer une génération. Elle est calculée en fonction du nombre de parties gagnées. Sur 10 parties jouées, un individu qui en gagne 6 aura donc une fitness de 6. Il aurait été possible d'accorder plus de poids à la fitness, en élaborant un système de puissance, par exemple : fitness $= \frac{2^{\text{nombre de parties gagnées}}}{Nombre\ Total\ de\ Parties}$. Ainsi, il est laissé la possibilité aux individus faibles de passer dans la génération suivante, mais le poids associé aux individus forts est beaucoup plus important.
Enfin, le facteur de mutation d'un individu a été défini arbitrairement à 0.05, et lorsque cette probabilité est atteinte, l'individu voit ses coefficients redéfinis aléatoirement.

9.4 Résultats

Les résultats de l'algorithme génétique sont disponibles dans la section « Statistiques ».

Quatrième partie

Statistiques

9.5 Victoires/Defaites

Après exécution de l'algorithme génétique, l'intelligence artificielle du programme est capable de battre une IA niveau 15/30 haut la main (53 à 11). Les scores se resserrent de plus en plus lorsqu'on augmente de niveau. Cependant, le programme n'est toujours pas capable de battre un niveau 19/30 (perdu à 17 à 47).
Le programme est cependant largement capable de détrôner un joueur débutant et surpasse aussi sans problèmes un joueur ayant des notions d'othello plus poussées. Peut être qu'une distinction plus importante des coefficients et une exécution de l'algorithme sur un plus grand nombre de générations pourrait pallier ce problème. Malheureusement, je ne dispose pas de suffisamment de temps pour en faire l'expérience.

9.6 Exécution d'une Partie

9.6.1 Durée

Après lancement de plusieurs parties et calcul de la durée d'exécution, on peut approximer le temps moyen de calcul d'une partie à 97 secondes pour une profondeur de Min-Max de 6. Cela inclut l'ajout du dernier coefficient concernant le pattern carré sur les coins et les calculs en nombres flottants. Au final, le rapport entre la force de l'intelligence artificielle (moyenne) et la vitesse d'exéctution (lente) ne donne pas de résultats extraordinaires.

9.6.2 Occupation Mémoire

Les calculs ci-dessous se basent sur l'occupation mémoire des entiers courts (2 octets), des flottants (4 octets) et des pointeurs (8 octets). On peut estimer l'occupation mémoire complète et maximale d'une partie à :

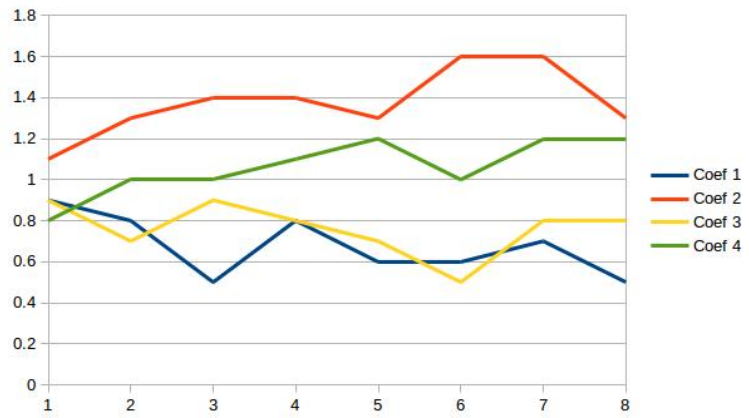
- Le tableau d'othelliers (estimé précédemment à moins de 13 Ko au maximum)
- Le tableau de coups (estimé à 480 octets octets au maximum)
- Les deux structures player (estimées à 17 octets chacune)
- Le tableau de coefficients (4 octets par coefficient soit 16 octets)
- Les différents curseurs (10 octets)
- L'utilisation mémoire de Min-Max, que l'on peut approximer en considérant 8 coups possibles par branche, soit 8^6 coups pour une profondeur de 6. Min-Max utilise la pile, soit $8^6 * utilisation\ de\ Min-Max$. Cette dernière valeur est estimée à 240 octets (par étage). Au total, on a une utilisation de min-Max de 60 Mo pour 6 étages et au maximum.
- Le pointeur vers la structure de jeu (8 octets)

L'utilisation maximale en mémoire et en même temps du programme est donc de 62,928,420 octets, soit moins de 61 Mo.

9.7 Algorithme Génétique

Après 7 jours de calculs (2 jours ont été perdus à cause d'une panne inopinée), l'algorithme génétique, bien que non optimal, a donné des résultats qui convergent : le meilleur génome de la dernière génération se trouve être la combinaison suivante : *0.5, 1.1, 0.7, 1.1*

Pour essayer d'apprécier l'évolution des coefficients choisis par l'algorithme génétique, voici un graphe représentant l'évolution des moyennes de chaque coefficient en fonction des générations :



Cinquième partie

Conclusion

9.8 Problèmes Rencontrés

Ce projet d'algorithmique a été, pour moi, extrèmement bénéfique, sous tous les aspects.
Premièrement, il m'a permis d'approfrondir encore ma connaissance du langage C, et m'a convaincu que ce langage est indispensable lorsqu'on veut privilégier le coté algorithmique d'un programme, tant par sa capacité de calcul que par sa gestion drastique de la mémoire.
De plus, ce projet m'a apporté beaucoup en matière de connaissances et m'a nourri d'une certaine curiosité concernant les algorithmes de décision, et plus généralement l'intelligence artificielle, domaine que je trouve fascinant.
Durant ce projet, j'ai rencontré quelques problèmes qui m'ont été bénéfique et m'ont appris, entre autre, à gérer la répartition de mon temps de travail. En effet, il est vite arrivé de se perdre dans la compréhension d'un algorithme, et le temps que j'ai passé à corriger des erreurs en décomposant les problèmes sur papier a été fatal à certaines améliorations que j'aurais aimé implémenter.
J'ai aussi appris, grâce à ce projet, que les évènements matériels imprévus sont nocifs au développement d'un programme, notamment concernant l'algorithme d'apprentissage génétique, qui aurait dû se terminer dans les temps avec la configuration prévue. De plus, l'importance d'un programme propre, modulaire et rapide a été soulignée lors de ce projet, en particulier pour le développement de l'intelligence artificielle. Je me suis rendu compte que, malgré la puissance de calcul actuelle de nos ordinateurs, certaines taches demandent beaucoup de ressources, et c'est pourquoi il faut veiller à les minimiser.

9.9 Améliorations non Effectuées

Voici une liste non exhaustive des améliorations possibles de ce programme que je n'ai pas eu le temps d'implémenter :

- Différents niveaux de jeu
- Deuxième algorithme de recherche de coup (pour comparer avec Min-Max)
- Stockage de l'arbre créé par Min-Max pour une plus grande rapidité d'éxécution
- Utilisation de tables de transposition (encore une fois pour améliorer la vitesse d'exécution)
- Utilisation du BitBoard (pour l'utilisation mémoire)
- Ajout de patterns propres à l'othello pour améliorer la fonction d'évaluation

Il y a bien sur plein d'autres améliorations à effectuer, et je regrette de ne pas avoir eu plus de temps pour finir d'améliorer mon prgramme.

9.10 Conclusion

En conclusion, ce projet m'a beaucoup apporté sur le plan technique (maitrise des algorithmes, implémentation en langage C, déboguage) ainsi que concernant l'organisation et la conception d'un projet complet (gestion du temps, programmation modulaire, recherche d'information).
Malgré les quelques petits problèmes mineurs rencontrés, je me suis épanoui tout au long de la réalisation de ce projet. Le seul regret que je pourrais exprimer est celui de ne pas avoir eu assez de temps pour finaliser et optimiser le programme, bien que les objectifs fixés par le cours aient été remplis.

9.11 Sources

- Fédération francaise d'Othello : http ://www.ffothello.org/
- Force des positions et fonction d'évaluation : http ://imagine.enpc.fr/ monasse/Info/Projets/2003/othello.pdf
- Page de Michael Buro : https ://skatgame.net/mburo/
- Strategies and tactics for intelligent search : http ://web.stanford.edu/ msirota/soco/minimax.html