Projet d'algorithmique

Licence 2 Informatique

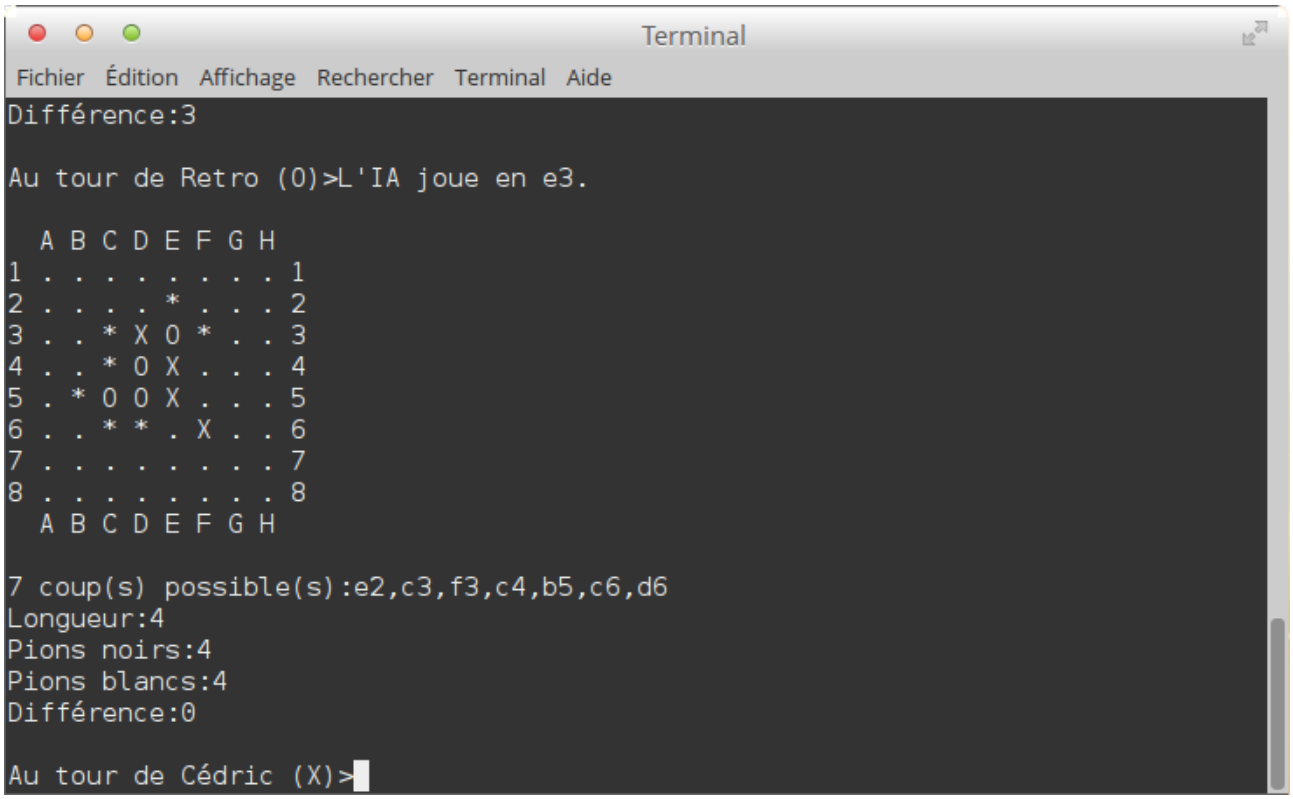
Université d'Aix-Marseille

20/04/13

Projet Reversi

Cédric Bérenger

berenger.cedric@gmail.com

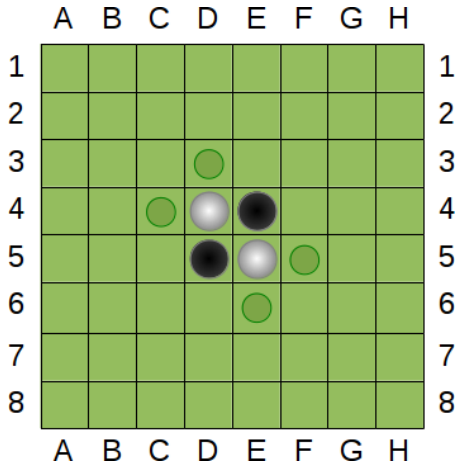


Aix*Marseille université

Table des matières

I Présentation du projet

I.1 Introduction



Le Réversi est un jeu de plateau pour deux joueurs et comme pour la plupart des jeux de plateau, il existe de nombreuses versions informatiques fournissant une intelligence artificielle plus ou moins développée, capable d'anticiper les réactions adverses avec plusieurs dizaines de coups d'avance en utilisant diverses techniques comme un arbre de recherche et un algorithme « min-max ».

Ainsi le but de ce projet universitaire est de mieux appréhender le fonctionnement d'une intelligence artificielle en réalisant un jeu de Réversi en langage C, qui, à terme sera « quasi imbattable » par un joueur humain, même expérimenté et capable rivaliser avec les meilleurs programmes du genre.

I.2 Règles du jeu

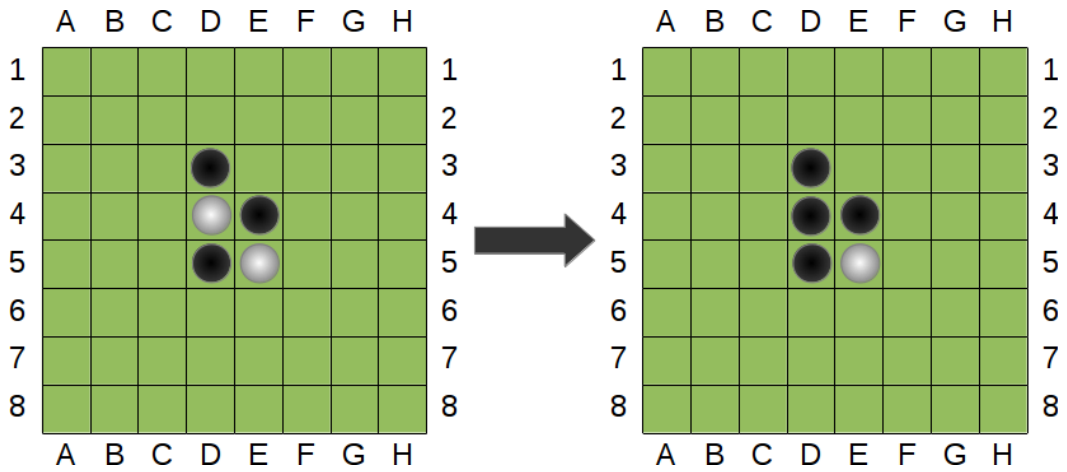
Le Réversi (aussi connu sous son appellation commerciale Othello) est un jeu de plateau qui se joue sur un damier carré de 64 cases avec des pions ayant une face noire et une face blanche.

Chaque joueur à sa couleur. Le joueur ayant les pions noirs commence. Au départ, il y a deux pions noirs et deux pions blancs sur le plateau.

Le joueur qui a la main doit poser un pion de manière à encadrer un ou plusieurs pions adverses avec un autre de ses pions déjà posé. Ainsi, les pions encadrés sont capturés et sont retournés de manière à prendre la couleur du joueur.

Les joueurs jouent ensuite à tour de rôle. S'il est impossible pour un joueur de placer un pion, il passe son tour.

La partie se termine lorsque le plateau est plein ou qu'aucun des deux joueurs ne peut plus jouer. Le gagnant est celui qui a le plus de pions de sa couleur sur le plateau.



I.3 Architecture du projet

Pour faciliter la programmation, le programme est découpé en six modules principaux, chacun réalisant une fonctionnalité spécifique :

- **Le module d'entrée-sortie « es »** regroupe les fonctionnalités relatives à l'affichage en mode textuel, les fonctions de sauvegarde et de chargement d'une partie ainsi que les fonctions de saisie et de conversion des données etc.
- **Le module de gestion d'othellier « othellier »** s'occupe de gérer le plateau de jeu : Initialisation, vérification de la légalité d'un coup, etc.
- **Le module de gestion de partie «partie »** gère le déroulement d'une partie (annulation d'un coup, navigation dans l'historique de la partie etc.)
- **Le module principal main** coordonne tous les modules et initie une partie.
- **Le module d'intelligence artificielle « ia »** regroupe les fonctions relatives à l'intelligence artificielle permettant à un humain de jouer contre l'ordinateur.
- **Le module d'apprentissage genetique « genetique »** permet d'améliorer le niveau de l'IA par apprentissage génétique.

A chaque module correspond concrètement un **fichier « .c »** ainsi qu'un **fichier d'en- tête « .h »** du même nom que le module. **Les fichiers « .h »** contiennent les définitions des structures de données.

La compilation des différents module en un seul exécutable « reversi » est quand à elle simplifiée par l'adjonction d'un fichier **makefile**.

II Les structures de données

II.1 Introduction

Avant même de commencer la programmation du jeu de Reversi, il faut prendre en considération qu'un mauvais choix de structure peut favoriser un code illisible ainsi que des erreurs de segmentations en C, ce qui rendrait tout débogage fastidieux. De plus, il serait difficile de faire un programme optimisé si les structures n'étaient pas réfléchies dans cette optique, Or, l'optimisation est cruciale pour notre projet puisque nos objectifs sont, à terme de créer une « IA » performante.

Ainsi, pour faciliter la réalisation du programme, il est nécessaire de choisir les bonnes structures données pour représenter les différentes composantes du jeu.

On peut choisir de découper le jeu de réversi en quatre composantes principales. En effet, une **partie de Réversi** se joue sur un **plateau de jeu, l'othellier**, sur lequel s'affronte **deux joueurs** qui jouent un **coup** chacun leur tour.

Enfin, pour une meilleure organisation, je choisis de recourir aux structures C qui permettent de classer des variables dans une arborescence, ainsi, je peux créer une structure en C pour représenter chaque composante du jeu et même assembler plusieurs structures pour en créer une seule grande (pratique pour simplifier les passages de paramètres).

Note : Dans les précédentes version de ce rapport, je présentait des structures de données assez « classique », depuis, j'ai opté pour des structures moins gourmandes en mémoire et donc plus rapide lors des opérations de recopies : le BitBoard.

II.2 Structure othellier

Pour représenter l'état du plateau de jeu (ou **othellier**), je choisis de créer une structure C othellier pour contenir :

- le contenu de chaque case du plateau,
- le nombre de pions pris par chaque joueur,.

```
typedef struct{
    unsigned long long presence;
    unsigned long long couleur;
    int materiel[2];
} othellier;
```

II.2.a Tableau damier

Pour sauvegarder le contenu de chaque case de l'othellier, j'utilise deux entiers de 64bits :

- Dans le premier entier, chaque bit code la présence d'un pion sur chacune des 64 cases de l'othellier.
- Dans le deuxième, chaque bit code la couleur des pions posés sur chaque cases de l'othellier. (Si il n'y a pas de pions sur la case, on ne soucie pas de la valeur du bit).

	A	B	C	D	E	F	G	H	
1	0	1	2	3	4	5	6	7	1
2	8	9	10	11	12	13	14	15	2
3	16	17	18	19	20	21	22	23	3
4	24	25	26	**27**	**28**	29	30	31	4
5	32	33	34	**35**	**36**	37	38	39	5
6	40	41	42	43	44	45	46	47	6
7	48	49	50	51	52	53	54	55	7
8	56	57	58	59	60	61	62	63	8
	A	B	C	D	E	F	G	H	

Cette structure a l'avantage d'utiliser un minimum de mémoire, de plus, comme les processeurs actuels fonctionnent sur 64bits, les opérations de recopies sont très rapides, on peut donc sans hésiter sauvegarder un othellier entier, ce qui sera fortement utile par la suite.

Note : Cette structure de donnée à cependant un désavantage par rapport à l'ancienne: Il n'est pas possible de faire des « bords » au tour de la surface de jeu pour éviter de regarder en dehors du tableau lors d'un test de légalité d'un coup par exemple, il faut alors trouver un autre moyen pour empêcher cela.

II.2.b BitBoard: Gestion du damier

L'utilisation d'une structure en bitboard pour l'othellier conduit à une gestion particulière du damier puisqu'il faut manipuler des bits et non des entiers. Il faut donc, pour l'implémentation en C connaître la manière dont se manipulent les bits en C.

Exemple d'othellier sauvegardé sur 2*64bits (Il s'agit de l'othellier de début de partie):

Position :	63	62	61	…	36	35	…	28	27	…	2	1	0
Présence :	0	0	0	…	1	1	…	1	1	…	0	0	0
Couleur :	0	0	0	…	1	0	…	0	1	…	0	0	0

b.i Accès à une case du damier

Pour accéder au contenu de la $i^{ème}$ case du damier, il faut connaître le $i^{ème}$ bit de l'entier codant la présence de pions ainsi que le $i^{ème}$ bit de l'entier codant la couleur du pions.

On peut par exemple faire deux fonction: Une qui retourne s'il y a un pion sur la case, et une autre qui retourne la couleur de ce pion.

Ces deux fonctions s'implémentent en C comme ceci:

```
int couleur_pion(othellier * o, int pos){
        return (o->couleur & ((unsigned long long)1<<pos))>0;
}
int presence_pion(othellier * o, int pos){
        return (o->presence & ((unsigned long long)1<<pos))>0;
}
```

L'opération "1<<pos" permet d'obtenir un entier dont le bit de position pos est à un, le reste à 0.

Ensuite, on réalise un "ET" bit à bit ("&") avec l'entier codant la présence ou la couleur d'un pion sur une case, de cette manière, si il n'y à pas de pions, le résultat sera 0, sinon, un nombre positif.

b.ii Modification d'une case du damier

Pour modifier le contenu d'une case, on crée deux fonctions, une pour retourner un pion sur une case, une autre pour poser un pion:

```
void retourner_pion(othellier * o, int pos)
{
        o->couleur^=((unsigned long long)1<<pos);
        o->materiel[couleur_pion(o,pos)]++;
        o->materiel[!couleur_pion(o,pos)]--;
}
```

Ici, comme pour l'accès à une case, on effectue l'opération "1<<pos" de manière à avoir un entier dont le bit de position pos est à 1.

On réalise alors l'opération "Ou Exclusif" ("^") avec l'entier de la structure othellier codant la couleur des pions, ce qui aura pour effet d'inverser le bit de position pos et donc de retourner le pion.

Ensuite, on met à jour le nombre de pions de chaque joueurs.

```
void poser_pion(othellier * o,int pos, int c) {
o->presence|=((unsigned long long)1<<pos);
	if(c)
	o->couleur|=((unsigned long long)1<<pos);
	else
	o->couleur&=~((unsigned long long)1<<pos);
o->materiel[c]++; }
```

Lorsque l'on veut poser un pion, on met le bit de position pos à 1 dans l'entier codant la présence grâce à l'opérateur "OU" bit à bit ("|").

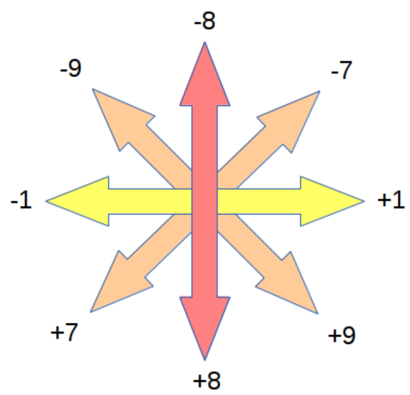
Ensuite:

- Si la couleur du pion est Blanc, c vaut 1 et donc on met le bit de position pos à 1 comme pour la présence.
- Sinon, c vaut 0 et donc on met le bit de position pos à 0 de la maniere suivante: l'opérateur "~" réalise un "NON" bit à bit, ce qui permet d'obtenir un entier dont le bit de position pos est à 0, les autres à 1. On réalise alors un "ET" logique bit à bit avec l'entier codant la couleur des pions.

Remarque: par défaut, sur certaines plate-formes, lorsque l'on effectue l'opération de décalage "1<<pos", le C la fait mais sur 32 bits, ce qui est problématique. Il faut donc « caster » l'opération et écrire « (unsigned long long)1<<pos ».

II.2.c Déplacement dans le damier

Il est facile de ce déplacer dans le damier par rapport à une position p. En effet, pour se déplacer verticalement, il suffit, comme on est sur un plateau de 64 cases (sans bords), d'ajouter ou de retrancher 8. De même, pour se déplacer horizontalement, if suffit d'ajouter ou de retrancher 1.



Comme évoqué précédemment et comme il n'y a pas de bords, il faut vérifier que l'on ne regarde pas en dehors du damier. Pour cela, j'opte pour la création d'un tableau auxiliaire pour indiquer lorsque l'on sort du damier :

1	1	1	1	1	1	1	1	1	1
1	0	0	0	0	0	0	0	0	1
1	0	0	0	0	0	0	0	0	1
1	0	0	0	0	0	0	0	0	1
1	0	0	0	0	0	0	0	0	1
1	0	0	0	0	0	0	0	0	1
1	0	0	0	0	0	0	0	0	1
1	0	0	0	0	0	0	0	0	1
1	0	0	0	0	0	0	0	0	1
1	1	1	1	1	1	1	1	1	1

Ainsi, pour vérifier si une position est valide après un déplacement, il suffit accéder à la case de coordonnées (pos %8 ;pos/8).

Note : Pour optimiser le programme, on remplacera lors de l'implémentation les opérations de divisions par un décalage des bits à droite (Ex : « >>3 ») et les opérations de modulo en utilisant un masque pour récupérer les bits de poids faible (Ex : « &7 »).

II.2.d Nombre de pions

Les nombres de pions pris par chaque joueurs sont simplement stockées dans deux entiers. Ils serons mis à jours lorsqu'un des deux joueurs joue, ce qui permettra notamment d'éviter de scanner la totalité du plateau de jeu pour connaître le gagnant.

II.3 Structure coup

Un coup est simplement constitué deux entier :

- Une position où un joueur décide de placer un pion.
- La couleur du joueur effectuant le coup.

Note : Contrairement à la précédente version du rapport, il n'y a pas de tableau de retournement, en effet, le retournement des pions sera directement effectué sur l'othellier lors du test de légalité d'un coup. De plus , et puisqu'il ne faut que deux entiers pour faire un coup, je ne crée pas de structure C dédiée lors de l'implémentation.

II.4 Structure partie

Enfin, pour organiser les variables relatives à la partie jouée et pour éviter d'avoir à rentrer de nombreux paramètres lors des appels aux fonctions, je choisis de créer une structure C **partie,** combinaison de plusieurs structures :

- **La structure othellier** qui pour rappel contient l'état du plateau de jeu (nombre de pions et contenu de chaque cases),
- **La structure joueur** qui contient simplement diverses informations relatives aux joueurs (leur nom, si ils sont réels ou des intelligences artificielles etc.)
- **La structure historique** qui contient une sauvegarde de chaque othellier après chaque coup joué, ce qui permet de naviguer facilement dans la partie.
- À cela s'ajoute quelques variables permettant de connaître l'état de la partie :
  - Le **trait**, qui est la couleur du joueur qui à la main et qui est donc sur le point de jouer.
  - La **longueur** de la partie qui indique ou se situe la partie c'est à dire le nombre de coups déjà joués.
  - La **longueur_max** de la partie est utilisée pour permettre des « retours avant » dans la partie : En effet , lorsque l'on recule dans la partie on décrémente la longueur, mais si l'on veut annuler le retour, il faut savoir si l'on a le droit d'avancer pour ne pas accéder à un coup qui n'a pas encore été joué. C'est le but de cette variable qui contient la longueur maximale de la partie. Il faut bien sûr penser à l'écraser si un joueur joue un coup.

```
typedef struct{
        char nom[16];
        int type;
} joueur;

typedef struct{
        int trait;
        othellier o;
} historique;

typedef struct{
        int longueurmax;
        int longueur;
        int trait;
        othellier o;
        joueur j[2];
        historique h[64];
} partie;
```

III Interface et gestion de partie

III.1 Introduction

Comme évoqué précédemment, la première étape concernant la réalisation du jeu est de programmer les fondements pour ensuite pouvoir se concentrer pleinement sur l'Intelligence artificielle. À la fin de cette partie, le programme sera à même de gérer le déroulement d'une partie humain contre humain.

III.2 Module Othellier othellier.c

Le module othellier contient les fonctions qui interagissent directement avec le plateau de jeu. Ainsi il prend notamment en charge l'initialisation du plateau de jeu, les tests de validité d'un coup, le retournement des pions etc. Il contient aussi les fonctions de gestions du damier que l'on a vue lors de la partie précédente sur les structures de données.

III.3 Fonction d'initialisation initialiser_othellier(othellier * o)

L'initialisation de l'othellier est chose aisée avec La structure en BitBoard, en effet, l'othellier n'est en mémoire que deux entiers sur 64 bits. Ainsi, j'ai calculé les deux entiers qui correspondent à la position de départ : **103481868288** pour l'entier codant la présence, **68853694464** pour l'entier codant la couleur. Il suffit ensuite d'initialiser le matériel de chaque couleur à 2.

III.4 Fonction de test d'un coup légal (avec retournement de pions int legal_ret)

Cette fonction est la principale fonction utile au bon déroulement du jeu : elle teste si un coup est légal ou non et retourne vrai ou faux, (0/1 en C). Elle reçoit en paramètres **l'adresse de l'othellier**, **le coup à jouer** (une position et une couleur) ainsi que **l'adresse d'un second othellier**.

La fonction teste si la case choisie est vide et s'il est possible de capturer, c'est à dire de prendre en sandwich des pions adverses dans une des huit directions possibles.

La particularité de cette fonction est que, si le coup est valide, elle retourne un othellier où le coup aura déjà été joué, ainsi, il n'est pas nécessaire d'implémenter des fonctions pour jouer ou annuler un coup, on utilisera directement l'othellier retourné. (Pour annuler le coup, on naviguera dans l'historique).

Principe de base:

- Si la case proposée n'est pas vide, la fonction se termine et retourne que le coup est illégal.
- Sinon, pour chacune des huit directions :
  - On avance sur le damier dans la direction courante tant que la case contient un pion adverse.
  - Ensuite, si l'on tombe sur un pion de la couleur du joueur, le coup est alors légal et on retourne cette information que le coup est légal.
  - sinon on teste la direction suivante.
- Si dans aucune des huit directions il est possible de capturer des pions adverses, alors on renvoie que le coup est illégal.

Une astuce en C pour éviter de faire un bout de code différent pour chacune des directions consiste

à faire un tableau de huit éléments contenant les nombres à ajouter pour ce déplacer dans une direction. Pour plus de détail, Voir Annexe Module OTHELLIER othellier.c : Fonction de test d'un coup légal : Pseudo-code.

Attention ! Vu que la fonction « légal » doit renvoyer un othellier où le coup aura été joué, la fonction doit regarder obligatoirement toutes les directions.

Remarque : La fonction « légal » servira à la réalisation des fonctions permettant de savoir si un joueur peut jouer ou si la partie se termine.

III.5 Module partie partie.c

Le module partie contient les fonctions de gestions de la partie permettant entre autre la navigation dans une partie, c'est à dire revenir en arrière en annulant un coup, ou encore avancer dans la partie (annuler l'annulation). Cette navigation est entièrement basée sur la structure historique.

Rappel : Une **partie** (je parle ici de la structure) contient une « sous-structure » **historique** qui contient l'othellier initial et l'ensemble des othelliers résultants de chaque coups avec le trait correspondant (la couleur du joueur ayant la main). stockés dans un tableau de taille 60.

Ainsi, pour atteindre le début de partie, il suffit de mettre la variable **longueur** à 0 et de charger l'othellier stocké à la case 0 du tableau.
De même, pour reculer dans la partie en annulant le dernier coup joué, il suffit de décrémenter la longueur jusqu'à arriver à 0 ou tomber sur un othellier dont le trait correspondant est la couleur du joueur.

Idem pour avancer dans la partie, on incrémente la longueur en vérifiant que l'on arrive pas à al fin de l'historique (longueur > longueur_max).

III.6 Module E/S es.c

Le module d'entrée / sortie contient l'ensemble des fonctions permettant d’interagir avec le ou les utilisateurs. Il comprend la gestion de l'affichage, la sauvegarde et le chargement d'une partie à partir d'un fichier ainsi que diverses fonctions comme la gestion des commandes et le traitement des saisies utilisateurs.

III.6.a Sauvegarde d'une partie

Pour sauvegarder une partie, on profite de la simplicité de la structure en BitBoard de l'othellier et on stocke entièrement le contenu de l'historique dans un fichier.

III.6.b Chargement d'une partie

Le chargement fait l’opposé de l'opération de sauvegarde : On copie le contenu du fichier de sauvegarde dans l'historique.

III.7 Compilation et Mode d'emploi

La compilation d'un exécutable se fait à l'aide d'un makefile via la commande make. Le programme fonctionne en mode textuel et les interactions avec les utilisateurs se font via un prompt et des commandes explicites comme « d3 » pour jouer en d3, « sauvegarde NOM_FICHIER» ou « sav NOM_FICHIER» pour la sauvegarde par exemple.

La liste de ces commandes et leurs usages sont consultables In-game via la commande **aide** :

```
Commandes [PARAM1 PARAM2 ...]:  Description
=========================================
aide:                Affiche une page d'aide.
reculer:             Recule dans la partie juste avant le dernier coup joué.
avancer:             Action inverse de recule, avance dans la partie.
naviguer NUM:        Navigue dans la partie jusqu'à atteint le coup numéro NUM.
relancer:            Atteint le début de partie, un nouveau coup en commence
                     une nouvelle.
sauver NOM_FICHIER: Sauvegarde une partie dans le fichier NOM_FICHIER.
charger NOM_FICHER: Charge une partie sauvegardée dans le fichier NOM_FICHIER.
permuter:            Permute les joueurs.
quitter:             Quitte le jeu.
```

III.8 Transition vers la deuxième phase.

La première étape est finie, l'interface textuelle et la gestion d'une partie sont maintenant opérationnelles et fonctionnent « parfaitement » (du moins espérons que c'est le cas pour la poursuite du projet). Il n'y à donc pas d'améliorations notables à effectuer.

On peut maintenant se consacrer pleinement à la seconde étape qui est de commencer l'Intelligence Artificielle.

IV Création de l'intelligence artificielle

IV.1 Introduction

La seconde phase consiste à réaliser l'intelligence artificielle et ainsi permettre à un utilisateur de jouer seul contre l'ordinateur. Dans le cadre de ce projet, l'intelligence artificielle est basé sur un algorithme de type min-max.

Le principal avantage de ce type algorithme est qu'il choisi un coup avantageux pour l'IA en prenant en compte non seulement l'état actuel de l'othellier, mais aussi les futures réplique de l'adversaire et du joueur (l'IA) jusqu'à un nombre de coup donné, la profondeur, ce qui permet de gagner plus facilement.

La programmation de l'IA peut se décomposer en trois sous partie:

- Tout d'abord, il faut élaborer l'algorithme minmax qui constitue le cœur de l'intelligence artificielle.
- Ensuite, et comme la profondeur de recherche donnée à l'algorithme minmax est limité à cause du temps de calcul qui est exponentiel, il convient d'optimiser un peu l'algorithme. Ici on élaguera l'arbre de recherche en supprimant des branches inutiles grâce à un algorithme alpha-bêta, algorithme utilisé couramment avec le minmax.
- Finalement, l'algorithme minmax va de paire avec une fonction dite d'évaluation qui comme son nom l'indique évalue l'état de l'othellier afin de retourner un score relatif à la qualité d'un coup.

Attention ! Avant de se lancer dans la programmation de l'IA et commencer le MinMax, il faut impérativement être sur du bon fonctionnement du mode deux joueurs ainsi que de la gestion du plateau. La moindre erreur peut engendrer un comportement aléatoire de l'IA difficile à déboguer.

IV.2 Algorithme MinMax

L'algorithme minmax est le moteur de l'intelligence artificielle de ce projet. Plutôt que de se focaliser simplement sur le coup à jouer,le MinMax anticipe les coups suivant de manière à être dans une position avantageuse dans un nombre p de coups que l'on appellera la profondeur de recherche.

Il consiste à regarder **TOUS** les coups possibles jusqu'à la profondeur p donnée et à attribuer à chaque coup un score plus ou moins élever en considérant les avantages du coups pour le joueur ainsi que pour l'adversaire. Le meilleur coup est celui qui **MAXIMISE** les avantages pour le joueur et qui **MINIMISE** les avantages pour l'adversaire et donc le désavantage.

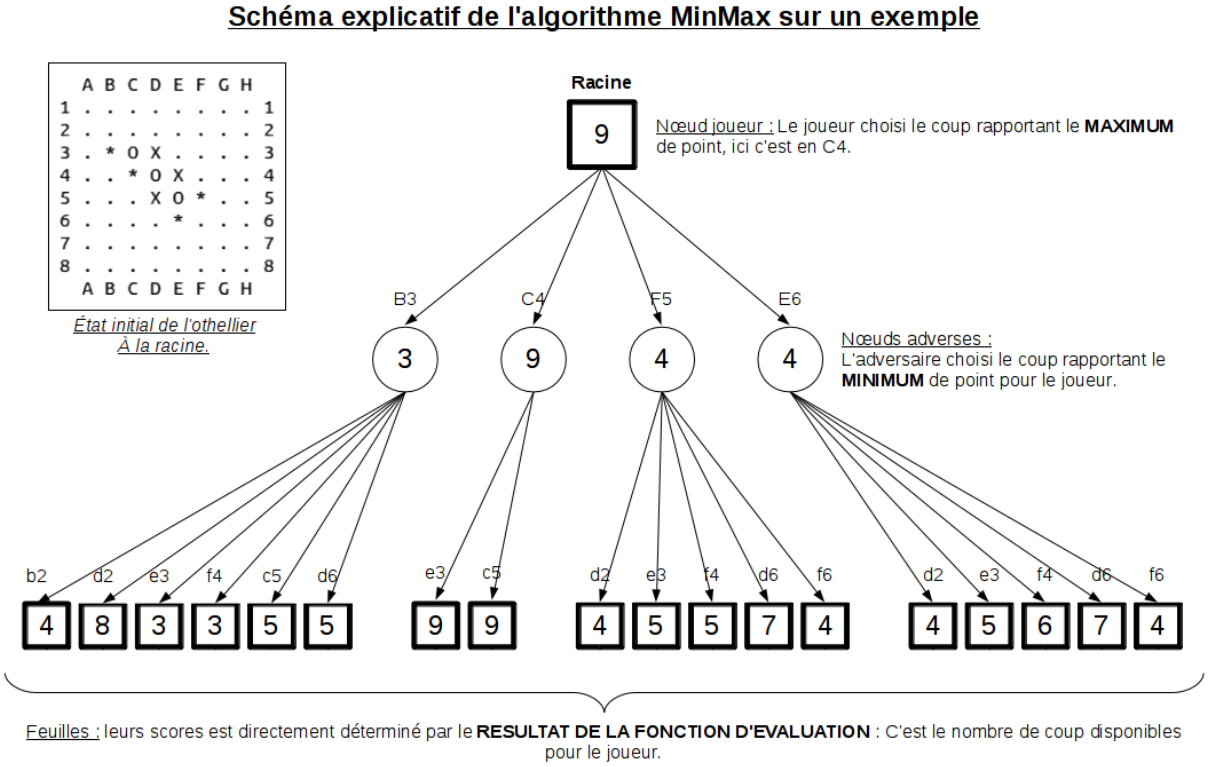
Principe:

Le MinMax explore l'arbre de tout les coups possibles jusqu'à la pondeur p et attribue un score à chaque nœud de l'arbre de la manière récursive suivante:

- Si le nœud est une feuille, c'est à dire si la profondeur maximale est atteinte ou que la partie se termine, alors le score est directement le résultat de la fonction d'évaluation.
- Sinon, si les fils du nœud sont des coups jouable par le joueur (l'IA), alors le nœud est un « nœud joueur » et son score est le maximum du score des fils.
- Sinon, les fils du nœuds sont des coups jouables par l'adversaire, le nœud est donc un

« nœud adverse » et son score est le minimum du score des fils.

Pour mieux comprendre, voici un schéma illustrant un exemple concret:



On considère que l'on est au tout début d'une partie et que le joueur (l'IA) à les noir et l'adversaire les blanc et que l'IA à déjà jouer en d3 et l'adversaire à répliqué en c3.

On considère aussi que la profondeur de recherche est de 2. (Je commence volontairement l'étude après deux coups car les premiers coups sont symétrique et donc les scores sont les même vu la faible profondeur de recherche.)

Comme nous somme encore qu'au tout début de la partie, le nombre de pions capturé importe peu, la mobilité, c'est à dire le nombre de coups qu'un joueur peut placé, quand à elle (et comme on le verra plus tard lorsque l'on abordera les aspects stratégique du jeu) est plus intéressante. En effet, si l'on a une bonne mobilité, il sera plus facile de placer des coups rapportant un maximum de points, à l'inverse, si l'adversaire à peu de choix concernant ses coups, il lui sera difficile de jouer correctement. Ainsi on prendra comme fonction d'évaluation le nombre de coups possibles.

IV.3 Implémentation en convention NégaMax

Lors de la réalisation de l'algorithme MinMax, on peut opter pour une implémentation en convention NégaMax qui a l'avantage d'être plus légère en terme d'écriture (Par contre, elle n'est pas forcément plus rapide).

Cette implémentation se base sur le fait que réaliser le minimum de plusieurs valeurs reviens à réaliser le maximum des valeurs opposées puis d'inverser le signe de la valeur retourné, ainsi il n'est plus nécessaire de considérer séparément les cas ou l'on se situe sur un nœud joueur ou un nœud adversaire.

IV.4 Élagage Alpha-Bêta

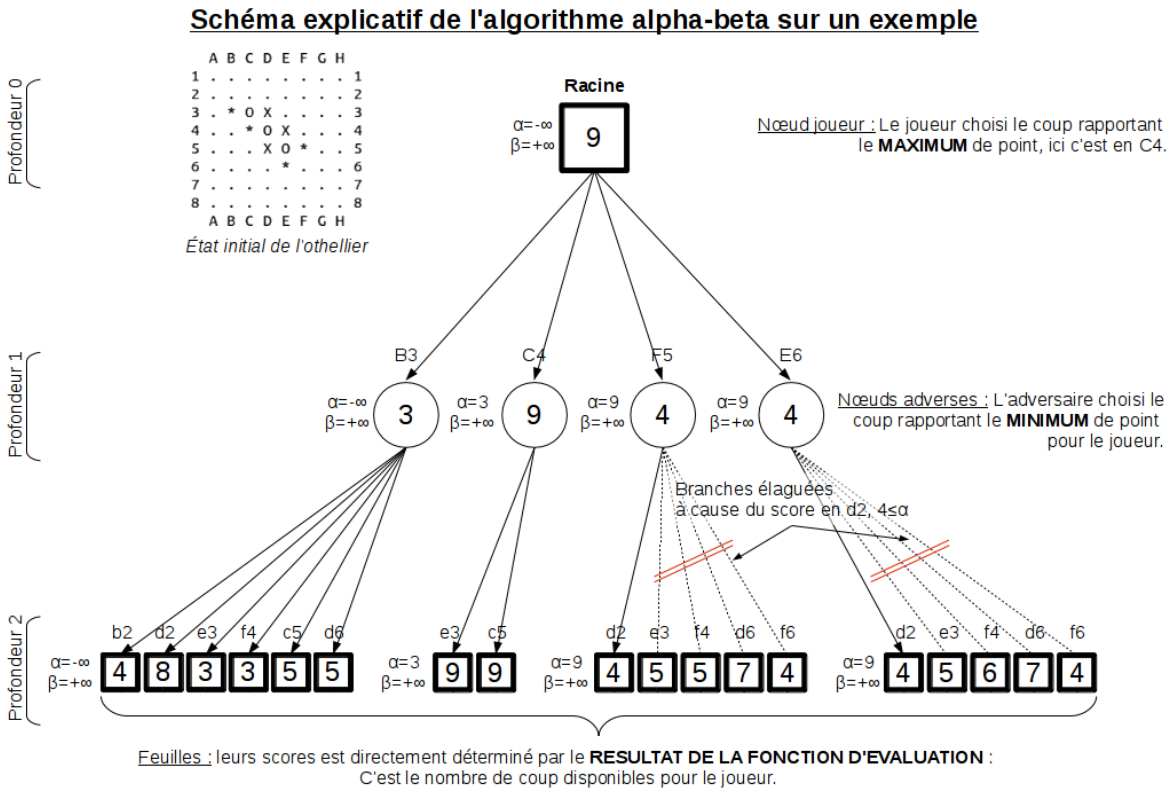
Comme dit précédemment, la profondeur de recherche est limité car le temps de calcul est exponentiel par rapport à la profondeur p, or plus la profondeur de recherche est grande, plus les chances de gagner augmente, d'où l'intérêt de supprimer certaines branches qui ne sont pas indispensables à l'arbre de recherche pour diminuer le temps de calcul. C'est l'objectif de l'algorithme alpha-bêta.

L'alpha-bêta est une simple optimisation du MinMax. Le principe est de restreinte la fenêtre de recherche en enlevant les branches dont on est certain que les nœuds aurons un score inférieurs à ceux déjà exploré.

Pour un niveau donné de l'arbre, l'algorithme crée deux variables, une **variable alpha** qui jouera le rôle d'une **borne min** qui contient **le score le plus élevé** parmi ceux des nœuds déjà visité, et une **variable bêta** qui à l'inverse joue le rôle d'un **borne max** qui contient **le score le plus faible** parmi ceux des nœuds déjà visités.

Ces deux variables forment alors un encadrement qui limite la fenêtre de recherche aux nœuds réellement indispensables, ainsi, lorsque l'on tombe sur un nœud au niveau suivant dont le score sort de l'encadrement, on arrête la recherche. Bien sur, on considère seulement l'une des deux bornes selon s'il on est dans un nœud joueur ou un nœud adversaire.

Voici un schéma explicatif sur le même exemple que pour le minmax original :



IV.5 La fonction d'évaluation

Le MinMax, ou encore son amélioration avec alpha-bêta, seul ne sert à rien, c'est la fonction d'évaluation qui est le véritable élément stratégique de l'intelligence artificielle.

Comme évoqué précédemment, elle renvoie un score qui jugera la qualité d'un coups passé en paramètre selon certains critères. Plus le score est efficace pour juger la qualité, plus l'IA sera efficace.

Explorons alors rapidement quelles pistes de fonctions d'évaluations simples, La partie dernière partie étant dédié à la recherche de fonctions d'évaluations efficaces et aux améliorations du MinMax/Alpha-Beta :

Considérons 3 critères primaires pour l'évaluation:

- **La différence du nombre** de pions entre le joueur et l'adversaire.
- **La différence de mobilité**, c'est à dire le nombre de coups qu'il est possible de jouer au tour suivant.
- **La différence du nombre de coins** possédés.

La fonction d'évaluation peut simplement retourner une combinaison linéaire des valeurs des 3 critères précédant. Cependant les coefficients de cette combinaison son très importants et sont fonctions de l'importance accordée au critère choisi. En effet, en début de partie, on privilégiera un coup permettant de capturer un coin qu'un coup permettant de capturer un grand nombre de pions adverses.

IV.6 Bilan de la phase 2, amorce de la phase 3

Après implémentation de l'algorithme alpha-bêta et d'une fonction d'évaluation rudimentaire, le programme est capable (étonnement) de battre AJAX en mode expert, l'objectif fixé au début du projet ce qui est encourageant pour la suite.

Cependant l'intelligence artificielle n'est pas parfaite loin de là et de nombreuses améliorations peuvent être réalisés en suivant plusieurs piste :

- Tout d'abord, on peut vouloir optimiser le cœur de l'IA, l’alpha-bêta dans le but d'en réduire le temps de calcul et permettre une recherche à une profondeur plus élevé.
- Ensuite on peut se pencher sur la fonction d'évaluation et essayer d'améliorer les scores retournés de manière à ce qu'ils soient plus représentatif de la situation de jeu en cherchant par exemple de nouveaux critères mieux adaptés.
- Enfin, on peut essayer d'implémenter des méthodes d’apprentissages automatiques, par exemple pour la génération des coefficients de la fonction d'évaluation.

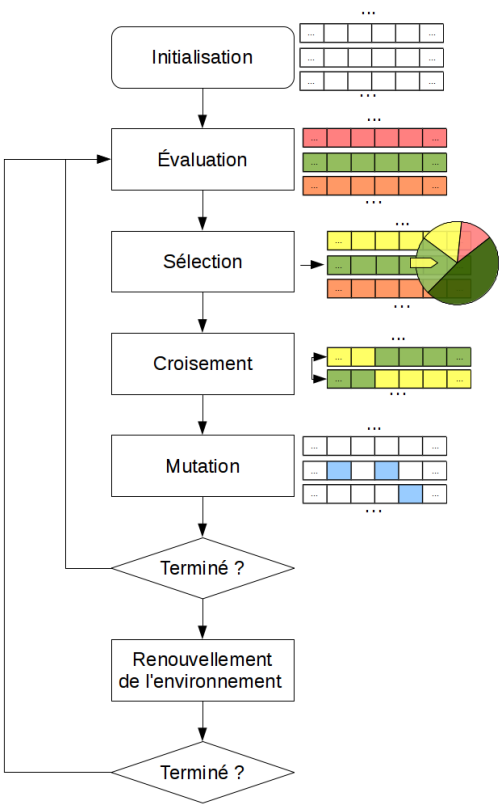
Le fonctionnement du programme demeure inchangé, si ce n'est que lors de la saisie du nom des joueur on doit choisir le type du joueur.

V Fonctions d'évaluations, algorithme génétique, et améliorations diverses.

V.1 Algorithme génétique

Pour générer automatiquement de meilleurs coefficients pour la fonction d'évaluation, on utilise un « **algorithme d'apprentissage génétique** ».

Son principe est simple et on peut y voir une analogie avec le monde de la biologie, la théorie de l'évolution de Darwin et de la sélection naturelle des espèces selon laquelle seul les individus les mieux adaptés à un environnement vont ce reproduire, évoluer au fil du temps et devenir de plus en plus adapté à leur milieu de vie, comme par exemple une bactérie qui avec le temps devient résistante à un médicament.



Ici, dans le cadre du jeu d'othello, on procède de la manière suivante :

- On crée une **Population** de n joueurs artificiels (IA). Chaque joueur est appelé un **Individu**. Chaque Individu de la Population utilise ses propres jeux de coefficients pour la fonction d'évaluation, on appelle ces coefficients le **Génome** d'un **Individu** et un seul coefficient est appelé un **gène**.
- On teste alors **l'adaptabilité** de chaque Individu, sa **« fitness »** en anglais, par rapport à des joueurs témoins, **l'environnement** en faisant s'affronter chaque individu de la population contre chaque Individu de l'Environnement. Le but de l'algorithme génétique est de générer de nouveaux individus qui, à terme seront capable de battre cet environnement et disposerons donc de coefficients optimaux que l'on récupérera.
- L'algorithme **sélectionne** alors les meilleurs individus.
- L'algorithme effectue ensuite un **croisement** des gènes des meilleurs individus.
- Puis effectue des **mutations** sur certains gènes de certains individus.
- On obtiens donc une **nouvelle population**, chaque nouvelle population généré est appelée une **génération**.
- On réitère ensuite ces étapes un certains nombre de fois jusqu'à l'obtention d'individus obtenant une assez grande fitness.
- On peut ensuite **renouveler l'environnement** en y mettant les meilleurs individus de la population et recommencer tout le processus, sachant que plus l'environnement est bon joueur, plus les individus générer seront de bons joueurs.

Attention ! Se lancer dans l'implémentation d'un algorithme génétique nécessite un Alpha-bêta fonctionnant parfaitement, si cette condition n'est pas remplie, attendez-vous à de longues et interminables heures de débogage !

V.1.a Calcul de fitness

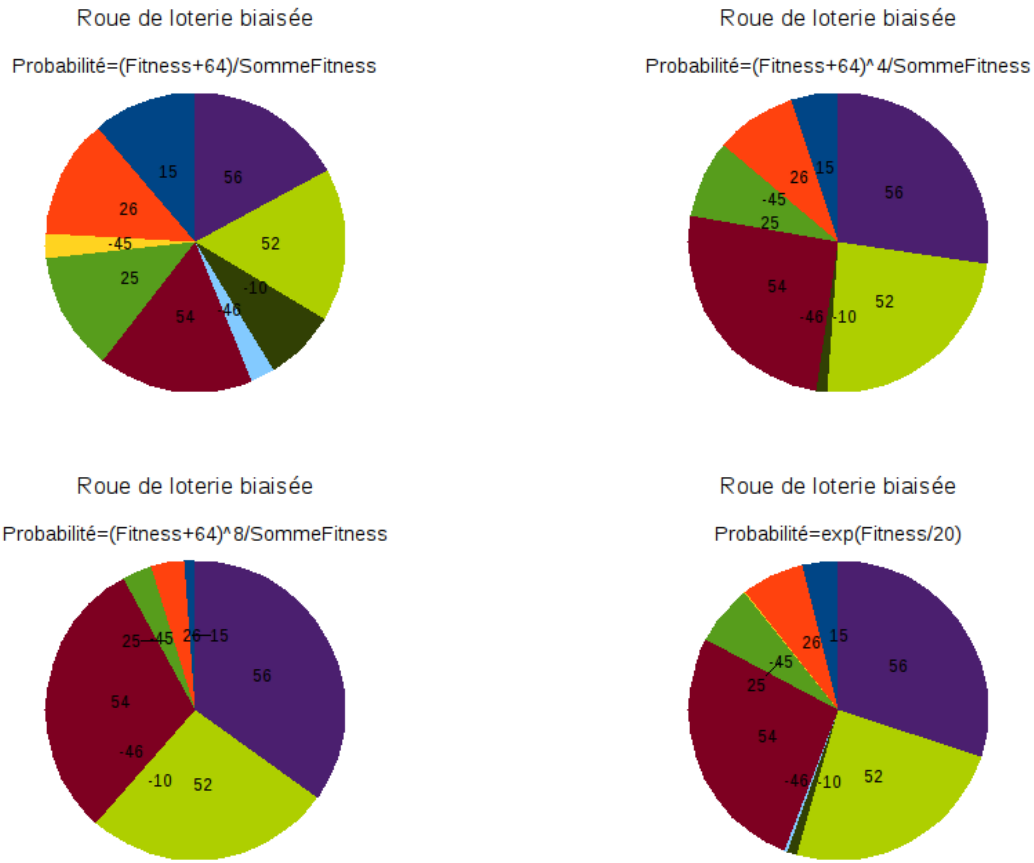
Dans le cadre du jeu d'othello, le calcul de la fitness se fait simplement de la façon suivante: On fait jouer chaque individus de la population contre chaque individu de l'environnement deux fois, une en noir, une en blanc, et on relève la différence du nombre de pions en fin de partie. Il suffit alors de faire la moyenne de ces nombres pour obtenir une valeur de fitness.

V.1.b Sélection des individus

Les individus sont sélectionnés de manière à privilégier ceux qui sont les plus adaptés, c'est à dire ceux ayant la plus grande valeur de fitness. Pour cela, on peut utiliser une roue de loterie biaisée: On s'arrange pour que chaque individu ait une probabilité d'être sélectionné proportionnelle à sa fitness.

Bien sûr, il ne faut pas oublier qu'une probabilité est comprise entre 0 et 1 et que la somme des probabilités doit faire 1, or ici, on la fitness peut être négative, il faut donc trouver une fonction permettant d'adapter la fitness obtenir un nombre remplissant ces deux critères.

Voici des exemples de fonctions pour la roue biaisée :



Pour ne traiter que des nombres positifs, je rend la fitness positive en y ajoutant 64 (-64 étant le score minimum pouvant être obtenu lors d'une partie), de plus, pour que la somme des probabilités fasse 1, je divise la fitness par la somme de toutes les fitness de la population.

On remarque sur ces diagrammes que s'il on choisi un calcul linéaire de la probabilité (diagramme circulaire 1), même si les fitness on un écart assez grand, par exemple 25 et 54, elles ont des probabilités similaires, Or, on veut évidemment privilégier fortement les fitness élevés comme 54. C'est pour cela que je choisi d'élever la fitness à une puissance. Élever à la puissance 4 semble être un bon compromis dans le sens ou cela privilégie les fortes valeurs mais conserve de la diversité en permettant à des valeurs plus faibles d'être sélectionnés.

Utiliser l'exponentielle ne semble pas être une bonne idée : D'une part, si les valeurs de fitness sont trop éloignée de 0, cela va privilégier un seul individu et les autres auront une probabilité d'être sélectionné proche de 0. D'autre part, s'il l'on « réduit » la fitness (ici je la divise par 20, voir diagramme 4) on obtient un résultat similaire à ceux obtenu en élevant à la puissance 4.

Ensuite, on réitère le processus de sélection jusqu'à obtenir une population de la même taille que la population initiale. Il est préférable de choisir de faire des population ayant un nombre pair d'individu pour avoir des couples pour effectuer l'opération de croisement.

V.1.c Croisement des gènes

Le processus de croisement permet à deux individus d'échanger des parties de leurs génomes dans l'espoir d'obtenir un nouvel individu mieux adapté à l'environnement, on parle alors de **Reproduction** dans la mesure ou un couple d'individu parents engendre deux enfants.

On peut implémenter cette étape de la manière suivante:

- Pour chaque couples individus, on choisi de faire ou pas un croisement selon une certaine probabilité.
- S'il y est décider d'un croisement, on choisi ce que l'on appelle un **point de croisement**. Et on échange par exemple les gènes dont l'indice est inférieur au point de croisement.

Processus de croisement :

Génome 1:	-6	-5	-6	-1	10	-5	-1	3	1	-2
Génome 2:	-7	5	-8	-9	-9	-1	-9	-10	-7	8

Sélection d'un point de croisement PC :

Le point de croisement est choisi au niveau du 4ème gène.

Génome 1:	-6	-5	-6	-1	10	-5	-1	3	1	-2
Génome 2:	-7	5	-8	-9	-9	-1	-9	-10	-7	8

Application du croisement :

Les gènes d'indices inférieurs au point de croisement sont échangés :

Nouveau génome 1:	-7	5	-8	-1	10	-5	-1	3	1	-2
Nouveau génome 2:	-6	-5	-6	-9	-9	-1	-9	-10	-7	8

Remarque : Lors de l'implémentation du programme, j'ai choisi une variante de ce processus

utilisant deux points de croisements. Le principe reste identique.

V.1.d Mutation des gènes

Pour éviter que la population ne « converge » et ne contienne plus que des individus identiques ainsi que pour la diversifier, on effectue aléatoirement des mutations sur certains gènes de certains individus:

- Pour chaque gène de chaque individu de la population on choisi d'effectuer une mutation selon une certaine probabilité (de préférence assez faible).
- Si il est décidé de faire une mutation sur un gène, on remplace la valeur du gène (ce qui correspond à un coefficient pour la fonction d'évaluation) par une valeur tirée aléatoirement.

Processus de mutation :

Génome :	-6	-5	-6	-1	10	-5	-1	3	1	-2

Sélection des gènes :

Génome :	-6	-5	-6	-1	10	-5	-1	3	1	-2

Mutation :

Nouveau génome :	-6	-5	-9	-1	10	3	-1	-6	1	-2

Les gènes d'indices 3, 6 et 8 ont mutés.

V.1.e Mode d'emploi du module génétique

Le module d'algorithme génétique se compile et s'exécute différemment du programme de jeu **reversi**.

Pour compiler le module génétique, il suffit d'exécuter la commande «**make genetique**» dans le dossier du projet contenant les sources.

Une fois compilé, il suffit de lancer l'exécutable « **genetique** ».

L'exécutable génère alors les coefficients. Ceux-ci sont sauvegardés dans un fichier « gens ».

V.2 Améliorations de la fonction d'évaluation

Nous avons déjà vu dans la partie précédente en quoi consistait la fonction d'évaluation et son importance pour l'intelligence artificielle.

Nous avons abordé les critères primaires que sont la **différence du nombre de pions**, la **différence de mobilité** ainsi que la **différence nombre de coins**, nous allons ici les aborder en profondeur et voir de nouveaux critères.

V.2.a Pourquoi effectuer une différence ?

Vous avez sans doute remarqué qu'a chaque fois, pour chaque critère, on n'utilise pas directement le nombre de pions possédés par un joueur, ni le nombre de coins mais on effectue la **différence**. Et bien ce choix part du constat suivant :

« Il est par exemple plus avantageux d'avoir plus de coins que l'adverse que simplement avoir des coins. On peut en effet avoir 1 coins, mais l'adversaire en posséder 3. De même pour la mobilité, il est par exemple préférable d'avoir plus de mobilité que l'adversaire. »

On peut même aller encore plus loin en partant du raisonnement suivant:

« Il est plus avantageux d'avoir 1 de mobilité et l'adversaire 0 ce qui l'oblige à passer son tour qu'une mobilité de 10 pour le joueur (l'ia) et 5 pour l'adversaire. »

Pour se faire, il suffit de diviser la différence de mobilité par la somme :

- (mob_joueur-mob_adv)/(mob_joueur+mob_adv)
- (1-0)/1=1
- (10-5)/15=1/3.

On peut appliquer cette méthode pour chaque critère.

V.2.b Découpage de la partie en phase

Lorsque l'on avance dans la partie, l'importance attribuée à chaque critère change : En début de partie, il est par exemple important d'avoir le minimum de pions alors qu'à la fin, il faut en avoir le maximum.

Il faut donc découper la partie en plusieurs phases et attribuer des coefficients différents pour chaque phases. Il faut cependant faire attention à ce qu'il n'y est pas de brusques sauts au niveau des scores lorsque l'on avance dans la partie et s'assurer d'une certaine homogénéité (surtout lorsque la partie se termine alors qu'il y a encore des cases libres sur l'othellier).

V.2.c Score en fonction de la position des pions

Il est évident que certaines cases sous plus avantageuses que d'autre pour y placer des pions, on a déjà vu ça avec les coins qui ont l'avantage d'être imprenables par l'ennemi. On peut généraliser ce critère à l'ensemble du damier en créant des zones rapportant plus ou moins de points , en voici un exemple :

	A	B	C	D	E	F	G	H	
1	0	1	2	3	4	5	6	7	1
2	8	9	10	11	12	13	14	15	2
3	16	17	18	19	20	21	22	23	3
4	24	25	26	27	28	29	30	31	4
5	32	33	34	35	36	37	38	39	5
6	40	41	42	43	44	45	46	47	6
7	48	49	50	51	52	53	54	55	7
8	56	57	58	59	60	61	62	63	8
	A	B	C	D	E	F	G	H	

Ici, on favorise les zones vertes au détriment des zones rouges et jaune. Il est en effet maladroit de placer un pion dans la zone rouge car cela permet à l'adversaire de prendre le coin. On peut cependant faire évoluer ces zones : en effet, si l'on dispose d'un coin, les pions voisin deviennent imprenables, on les appellent **pions définitifs**.

De plus, on peut aussi faire évoluer ces zones lorsque l'on avance dans la partie.

V.2.d Pions définitif

Avoir des pions définitifs est un atout majeur, car on sait qu'il ferons parti du score à la fin de la partie, plus on à de pions définitifs, plus on s'assure d'un score minimum.

V.2.e Pions frontières

Les pions frontières sont les pions ayant dans leurs voisinage au moins une case vide. Il permettent par exemple de se faire une idée de la mobilité : plus l'adversaire a une frontière importante (c'est à dire un grand nombre de cases vide autour de ses pions) , plus le joueur (l'IA) a des chances d'avoir une mobilité importante et inversement.

V.2.f Pions internes

Selon l'avancement de la partie, il peut être intéressant d'avoir des pions n'ayant aucune case vide au tour car ceux si sont plus difficilement prenable par l'ennemi et on plus de chance d'être définitifs. De plus, de tels pions sont susceptible d'augmenter la mobilité pour le joueur.

V.3 Améliorations de l'alpha-beta

V.3.a Table de transpositions

Lors de l'exécution de l'algorithme Alpha-Beta, il y a de forte chances pour des séquences de coups différentes amènent à des situation identiques, et donc que l'algorithme développe plusieurs fois des branches identiques.

En partant de ce constat, on peut avoir l'idée suivante qui est de sauvegarder chaque branche développée (ici on sauvegarde un othellier) dans une table ainsi que les scores correspondants et la profondeur à laquelle la branche a été développée. Ainsi, avant de développée une nouvelle branche, on regarde dans la table si on ne l'a pas déjà développée avant à une profondeur égale ou supérieure. Si c'est le cas, on se sert du score pré-calculé, sinon, on développe la branche.

Remarque : On n'utilise pas le score si la branche correspondante à été développée à une profondeur plus petite car moins précis.

Cependant, on se heurte au problème suivant: Les ordinateurs actuels ne dispose pas d'assez de mémoire pour créer un tableau capable de contenir toutes les branches possibles.

Une solution est d'utiliser une table de hachage: On crée un tableau de taille fixé et on associe à chaque branche une « valeur de hachage », c'est à dire un index correspondant à une case de la table.

Bien sur, comme il y a plus de branches possibles que de cases dans la table, deux branches distinctes peuvent avoir le même index: c'est ce que l'on appelle une collision. Il faut donc trouver une fonction permettant de générer des index de façon à minimiser le nombre de collision.

Avec une telle fonction, la table peut s'utiliser de la manière suivante:

- Lorsque l'on a développé une branche, on stocke trois informations dans la table: L'othellier, la profondeur, ainsi que le score correspondant.

  Remarque: On voit ici l'avantage d'avoir utiliser des structures en BitBoard: le stockage d'un othellier coûte un minimum de mémoire (deux entiers de 64 bits, soit 8 octets) et ne sollicite que peut de ressources.

- Ainsi, lorsque l'on veut développer une nouvelle branche, on calcule sa clé en fonction de l'othellier pour trouver la case correspondante dans la table. Si celle-ci n'est pas vide, on regarde si l'othellier stocké correspond à l'othellier courant, si oui, le score est utilisable. Sinon, il s'agit d'une collision et il faut développer la branche.

V.3.b Fonction de hachage

La manière dont sont générer les index est importante pour les tables de transpositions. En effet, si celle-ci prend du temps, elle ralentira l'alpha-bêta et donc rend inutile l'utilisation des tables de transpositions. De plus, il faut essayer que les othelliers sauvegardés soient répartie uniformément pour limiter le nombre collision pour maximiser le nombre d'élagages.

b.i Fonction de hachage basée sur le modulo

Puisque l'othellier est codé sur deux entiers de 64bits, pourquoi ne pas faire simple et générer un index en opérant directement sur ces entiers ?

Voici une manière de générer un index:

- On note **PRES** l'entier de 64bits codant la **présence** d'un pion.
- On note **COUL** l'entier de 64bits codant la **couleur** d'un pion.
- On note **TAILLE** la taille de la table de transpositions.
- Index = (PRES % (TAILLE) + COUL % (TAILLE)) % TAILLE

On obtient bien un index compris entre 0 et TAILLE-1 dont le calcul est assez rapide. Reste à savoir si cette méthode permet d'avoir une répartition sans trop de collision.

Note : Il faut éviter de prendre TAILLE une puissance de 2 car faire PRES % TAILLE reviendrai à prendre les n=TAILLE premiers bits de PRES.

b.ii Fonction de hachage de Zobrist

Une fonction de hachage éprouvée est celle de Zobrist. Couramment utilisé pour les tables de transpositions des jeux de plateaux comme les échecs ou encore le jeu de go, elle s'emploie comme ceci:

- On attribue au préalable à chaque état possible de chaque case du damier (case vide, pion noir, pion blanc) une valeur choisie aléatoirement. On pourra stocker ces valeurs dans un tableau par exemple.
- Ensuite, on relève les valeurs correspondant à l'état courant de chaque cases de l'othellier et on les combines avec l'opérateur logique bit à bit « OU-EXCLUSIF » de manière à obtenir un seul entier.
- Il ne reste plus qu'à effectuer un modulo sur cet entier pour obtenir un index.

VI Bilan et point sur l'avancement du programme

L'implémentation de l'algorithme génétique à été difficile et a nécessité de nombreuses heures de réflexions et de débogage. Il semble maintenant opérationnel. Cependant, n'ayant pas encore fini l'implémentation des tables de transpositions dans l'espoir de booster l'alpha-beta, la génération des coefficients est vraiment longue, très longue, et donc je ne dispose pas à ce jour d'un jeu de coefficient optimal. Ainsi, l'ancienne version du programme reversi (non basée sur des structures en BitBoard) est la plus performante en jeu. Pour rappel, cette ancienne version rempli les objectifs de battre AJAX en mode expert avec une moyenne de +5 pions. (Les scores variants à cause de l'aléatoire du programme AJAX).

Les deux versions (nouvelle avec BitBoard, ancienne sans BitBoard) fonctionnent globalement de la même manière, seul le nom de certaines commandes diffèrent. Utilisez la commande « aide » In-game pour afficher un descriptif.

Même si ce projet touche à sa fin, il y a encore bien d'autre pistes à explorer pour améliorer encore plus le niveau de jeu :

- On peut toujours chercher de nouveaux critères ou améliorer la vitesse de calcul des critères existants comme la mobilité qui est assez gourmande.
- Améliorer l'algorithme alpha-bêta en cherchant de nouvelles techniques d'élagages et améliorer la vitesse de calcul, notamment avec les techniques comme le Negascout/PVS (voir sur le site de la FFO : http://www.ffothello.org/info/algos.php) qui, malgré plusieurs tentatives d'implémentation n'a fait que ralentir l'algorithme alpha-bêta.
- Où même explorer d'autres approches que le MinMax pour l'intelligence artificielle.