Université de Provence - Aix-Marseille 1
Licence d'informatique

Rapport de projet : Othellix

Victor Martin, Lucas Texier et Gaëtan Champarnaud
<victormartinfr@gmail.com, lulutdu30@hotmail.fr, gaetan_champarnaud@hotmail.com>

Marseille, le 15 avril 2011

Table des matières

Introduction **2**

I Présentation d'Othello et établissement d'un algorithme performant pour gérer une intelligence artificielle **5**

1 Présentation d'Othello **6**
1.1 Les règles d'Othello . . . 6
1.1.1 Principes de base . . . 6
1.1.2 But du jeu . . . 6
1.1.3 Position de départ . . . 6
1.1.4 Pose d'un pion . . . 7
1.1.5 Fin de la partie . . . 7
1.1.6 Différences avec Reversi . . . 7
1.2 Quelques stratégies . . . 7
1.2.1 La stratégie positionelle . . . 7
1.2.2 La stratégie de mobilité . . . 8
1.2.3 La parité . . . 9
1.3 Conclusion . . . 9

2 Intelligence artificielle **10**
2.1 L'algorithme MinMax . . . 10
2.2 L'algorithme MinMax limité . . . 11
2.3 L'élagage AlphaBeta . . . 12

II La fonction d'évaluation **14**

3 Les différents critères **15**
3.1 Le critère de positionnement . . . 15
3.2 Le critère des coins . . . 16
3.3 Le critère de mobilité . . . 16
3.4 Le critère de parité . . . 18
3.5 Le critère du matériel . . . 18
La fonction d'évaluation . . . 18

4 Les algorithmes génétiques **19**
4.1 Introduction . . . 19
4.2 Présentation . . . 19
4.2.1 Les paramètres . . . 19

Les individus . . . 19
La population . . . 20
L'environnement . . . 20
Le fitness . . . 20
4.2.2 Les opérateurs . . . 20
La sélection . . . 20
Les croisements . . . 21
Les mutations . . . 21
4.2.3 Le déroulement de l'algorithme . . . 21
4.3 Les statistiques . . . 22
4.3.1 Les statistiques sur la fonction de Rosenbrock . . . 23
Variation du nombre d'individus . . . 24
Variation du nombre de générations avant maturité de la population . . . 25
Variation du rang de la population finale . . . 25
Variation du taux de croisement . . . 26
Variation du taux de mutation . . . 27
Variation du nombre des meilleurs individus sélectionnés directement . . . 28
Variation du nombre d'individus sélectionnées aléatoirement . . . 28
Conclusion . . . 29
4.3.2 Les statistiques sur les parties d'Othello . . . 29
Variation du nombre d'individus . . . 30
Variation du nombre de générations avant maturité de la population . . . 30
Variation du nombre d'individus dans l'environnement . . . 30
Variation du nombre d'individus de l'environnement renouvelés . . . 30
Variation du nombre du nombre de générations avant renouvellement de l'environnement . . . 30
Variation du taux de mutation . . . 30
Variation du taux de croisement . . . 30
Variation du nombre des meilleurs individus sélectionnés directement . . . 30

Introduction

Dans ce rapport nous allons vous présenter les étapes à suivre afin de réaliser une intelligence artificielle du jeu Othello répondant aux règlex de jeu de base (règles tirées du site http ://www.ffothello.org). Le programme sera réalisé en C et la règle imposée est que l'ordinateur ne doit pas dépasser 10 secondes dans le temps de calcul du coup à jouer.

Dans un premier temps nous présenterons le jeu d'Othello, les règles et les stratégies basiques qu'il est nécessaire de connaître pour un bon départ ainsi que les problèmes rencontrés au cours des étapes de recherche et de programmation. Nous verrons également les optimisations nécessaires pour gagner en temps de calcul.

Dans une deuxième partie nous verrons comment optimiser le jeu de l'ordinateur grâce à une fonction d'évaluation prenant en compte des coefficients ajusté avec un algorithme génétique. Nous détaillerons le fonctionnement de cet algorithme ainsi que la manière dont nous l'avons utilisé pour déterminer les coefficients.

Première partie

Présentation d'Othello et établissement d'un algorithme performant pour gérer une intelligence artificielle

Chapitre 1

Présentation d'Othello

1.1 Les règles d'Othello

1.1.1 Principes de base

Othello est un jeu de stratégie à deux joueurs : Noir et Blanc. Il se joue sur un plateau unicolore de 64 cases, 8 sur 8, appelé othellier. Ces joueurs disposent de 64 pions bicolores, noirs d'un côté et blancs de l'autre.

1.1.2 But du jeu

Le but est d'avoir plus de pions de sa couleur que l'adversaire à la fin de la partie. Celle-ci s'achève lorsque les des deux joueurs ne peuvent plus jouer de coup légal.

1.1.3 Position de départ

Au début de la partie, 2 pions de chaque couleur sont positionnés au centre du plateau comme le montre l'image ci-dessous.

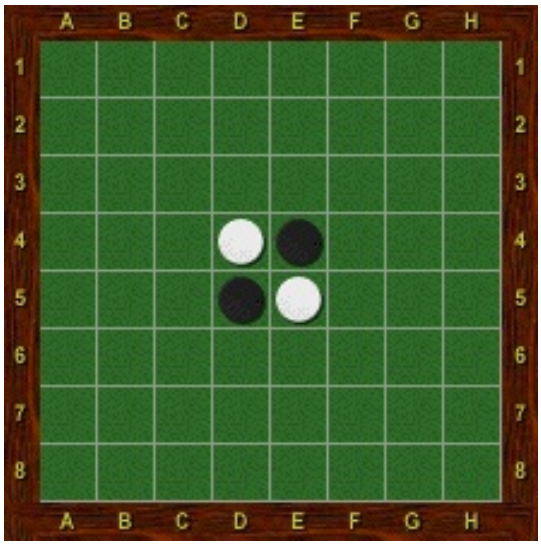

Figure 1.1 – l'othellier en début de partie

Par convention, le joueur noir commence la partie.

1.1.4 Pose d'un pion

Le joueur doit poser un pion de sa couleur sur une case vide de l'othellier, adjacente à un pion adverse. Il doit également, en posant son pion, encadrer un ou plusieurs pions adverses entre le pion qu'il pose et un pion à sa couleur, déjà placé sur l'othellier. Il retourne alors de sa couleur le ou les pions qu'il vient d'encadrer. Les pions ne sont ni retirés de l'othellier, ni déplacés d'une case à l'autre.

On peut encadrer des pions adverses dans les huit directions. Par ailleurs, dans chaque direction, plusieurs pions peuvent être encadrés. On doit alors tous les retourner.

Si, à votre tour de jeu, vous ne pouvez pas poser un pion en retournant un pion adverse suivant les règles, vous devez passer votre tour et c'est à votre adversaire de jouer.

1.1.5 Fin de la partie

La partie est terminée lorsque aucun des deux joueurs ne peut plus jouer. Cela arrive généralement lorsque les 64 cases sont occupées. Mais il se peut qu'il reste des cases vides où personne ne peut plus jouer : par exemple lorsque tous les pions deviennent d'une même couleur après un retournement.

On compte les pions pour déterminer le score. Les cases vides sont données au vainqueur. En cas d'égalité, elles sont réparties équitablement entre les deux joueurs.

1.1.6 Différences avec Reversi

Othello est en réalité une variante de Reversi, inventé presque un siècle auparavant (vers 1880). Ce jeu propose un scénario de départ différent : les joueurs posent, tour à tour, un pion sur l'une des cases centrales. Lorsque les 4 premiers pions sont posés, on peut donc aboutir à la même situation de départ qu'à Othello, mais il se peut aussi que chaque joueur ait une ligne, verticale ou horizontale.

Il est possible, contrairement à Othello, de passer son tour alors qu'on a accès à des coups légaux. Néanmoins, chaque joueur doit poser 32 pions au maximum, alors qu'à Othello il est possible de jouer plus de coups si l'adversaire doit passer. Si l'un des joueurs épuise ses 32 coups, l'autre a la liberté de poser ses derniers pions sur toutes les cases libres du plateau.

1.2 Quelques stratégies

Rappelons que le but du jeu est d'avoir plus de pions que l'autre à la fin de la partie et non pendant la partie. Heureusement pour l'intérêt du jeu, la stratégie qui consiste à retourner à chaque coup le plus grand nombre de pions possible est la plus mauvaise, comme nous allons le voir.

Plusieurs stratégies ont vu le jour, nous allons essayer d'en retenir les points forts afin de maximiser notre programme de jeu.

1.2.1 La stratégie positionelle

Cette stratégie consiste a repérer quels sont les pions définitifs. Un pion est dit définitif quand il ne peut plus changer de couleur jusqu'à la fin de la partie. Par exemple un coin est un pion définitif puisqu'il ne peut pas être pris entre deux pions de la couleur adverse.

Il ne faut donc pas donner inconsidérément des coins à votre adversaire. Le moyen le plus simple d'éviter de perdre un coin est de ne pas jouer sur les cases adjacentes au coin, que l'on appelle

cases X pour celles qui sont sur les diagonales a1-h8 et a8-h1 et cases C pour celles qui sont sur les bords.

Attention : Les bords ne sont pas des cases définitives et ne sont pas toujours bons à prendre, on peut souvent exploiter les faiblesses des bords adverses. Par exemple, dans le cas suivant :

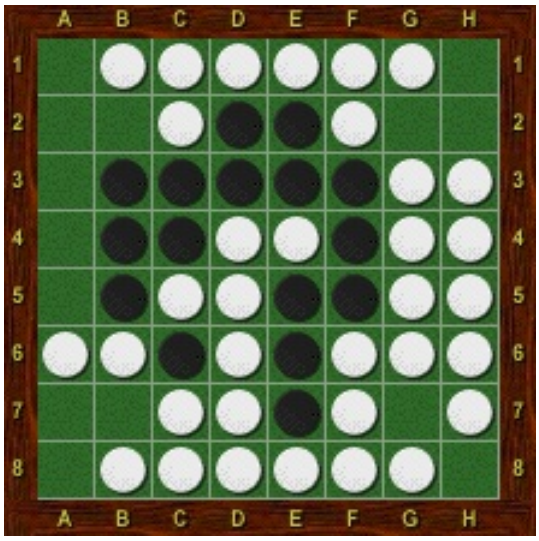

FIGURE 1.2 – un mauvais positionnement des bords

Blanc a beaucoup de bords et surtout, il a un "bord de cinq" sur la colonne h. Noir peut exploiter cette faiblesse en jouant la case g2. Certes, il perd alors d'emblée un coin si Blanc se précipite pour prendre h1, mais Noir peut alors s'insérer en h2 et gagner h8 au coup suivant (puis le coin a8). Blanc aura donc assuré 7 pions définitifs sur le bord nord, mais Noir en aura beaucoup plus à l'est comme au sud et gagnera la partie.

Il est donc important pour le programme de repérer et de prendre en compte les cases dites définitives dans les calculs Min-Max.

1.2.2 La stratégie de mobilité

Cette stratégie consiste à forcer l'adversaire à jouer un mauvais coup (par exemple à donner un coin sans compensation). Pour ce faire il est important de posséder le moins de pion possible dans l'objectif de ne donner qu'un petit nombre de coup possible a l'adversaire. En effet moins l'adversaire a de coup possible moins il a de possibilité de jouer un bon coup.

En général, plus vous placez de pions en frontière de la position, c'est-à-dire adjacents à des cases vides, plus votre adversaire a de coups et moins vous en avez.

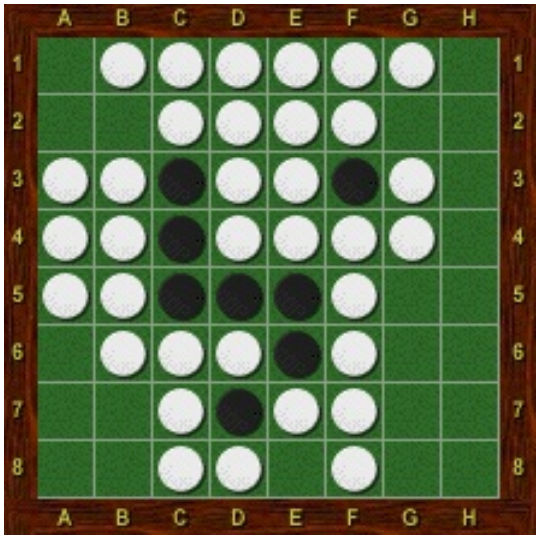

FIGURE 1.3 – un exemple de faible mobilité pour Blanc

Ici la mobilité du joueur blanc est très faible, en effet il n'a que deux coups possibles : b2, qui donne à Noir le coin a1 ou g2 qui donne le coin h1. De plus, comme Blanc a six pions sur le bord nord, dès que Noir prendra l'un des coins a1 ou h1, il pourra prendre l'autre au coup suivant. Noir a donc une très bonne mobilité.

1.2.3 La parité

Si aucun joueur ne passe son tour durant la partie, le joueur blanc joue le dernier coup de la partie et possède donc un avantage, en effet les pions retournés au dernier coup sont forcément définitifs. Le joueur noir a donc intérêt à essayé de faire passer le joueur blanc car le dernier coup est très important.

L'ordinateur doit donc prendre ce paramètre en compte. Pour cela, il doit essayer de faire en sorte que le nombre de tour passés soit s'il commence et impair s'il ne commence pas. Pour cela, il est parfois nécessaire de sacrifier un coup.

1.3 Conclusion

De ces stratégies nous pouvons tirer les principaux éléments nécessaires afin d'évaluer l'importance d'un coup à jouer et donc les principaux algorithmes à mettre en place :

- Les pions définitifs
- La mobilité
- La parité

Une fois ces algorithmes implémentés, il est important de connaître leur importance. Par exemple, si un coup permet de prendre deux pions définitifs, est-il mieux qu'un coup réduisant la mobilité de l'adversaire ? De plus, les réponses à ces questions sont différentes suivant l'avancement de la partie, au début la réponse peut être positive alors que dans le milieu elle serait négative.

Ces cœfficients peuvent être choisis aléatoirement dans un premier temps mais pour optimiser la réflexion de l'ordinateur, nous utiliserons un algorithme génétique afin de calculer, suivant l'avancement de la partie, des coefficients plus précis et mieux adaptés.

Chapitre

2 Intelligence artificielle

Lors de l'élaboration d'un jeu de plateau informatique, la question qui se pose est : comment faire jouer l'ordinateur ? Effectivement, parmi une liste de coups possibles, comment savoir quel coup choisir ? On pourrait choisir le coup qui rapporte le plus de points mais ce coup pourrait nous en faire perdre beaucoup par la suite. Il faut donc un algorithme qui choisit un coup de façon à ce qu'il soit profitable pour la suite de la partie, comme un investissement à long terme. L'algorithme MinMax est un algorithme qui permet de faire un tel choix pour les jeux dits "jeux à deux joueurs à somme nulle et information complète" (Othello est un jeu de ce type).

2.1 L'algorithme MinMax

Le but de l'algorithme MinMax est de trouver le meilleur coup à jouer à partir d'un état donné du jeu. Pour chaque coup que l'ordinateur peut jouer, l'algorithme va chercher les répliques que pourrait jouer le joueur adverse ; et pour chaque réplique, l'ordinateur va lister les coups qu'il peut jouer après ce coup adverse. Ainsi de suite jusqu'à atteindre la fin de la partie, qui finira par une victoire, une défaite ou une égalité. L'ordinateur choisira donc le coup qui le mènera vers la plus grande probabilité de victoire. Dans le situation de départ, l'ordinateur cherche à **max**imiser ses gains, il va donc choisir le coup qui l'amène vers la situation la plus profitable. Le joueur adverse fait de même et cherche à maximiser ses gains (donc **min**imiser ceux de de l'ordinateur). Voilà pourquoi l'algorithme s'appelle MinMax (ou encore MiniMax).

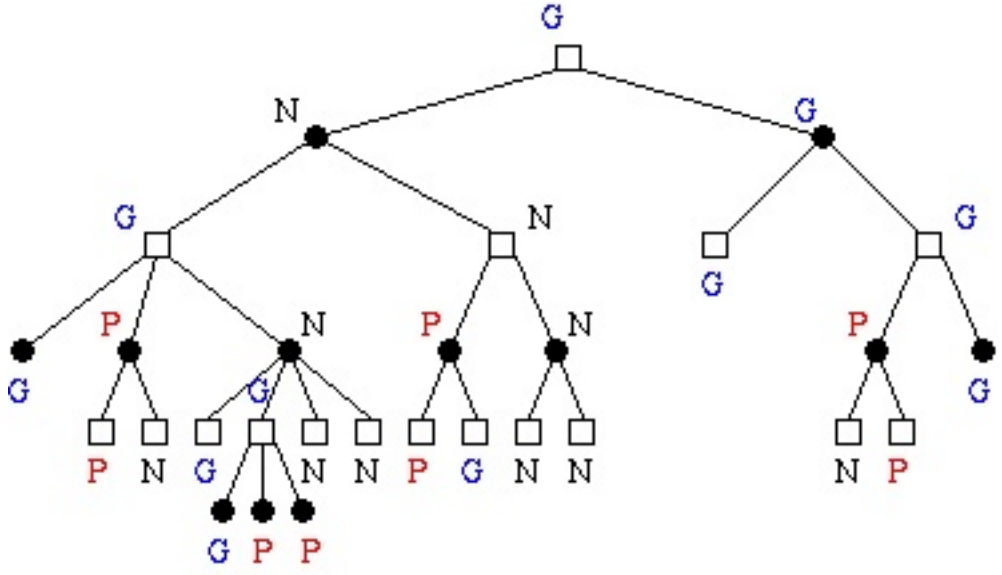

Figure 2.1 – Représentation des simulations de coups par l'ordinateur sous forme d'arbre

Dans cette image, chaque carré représente une situation de la partie où c'est à l'ordinateur de jouer, et chaque rond lorsque c'est au joueur adverse de jouer. Chaque branche représente un coup possible. Ici, l'ordinateur va choisir la branche de droite qui le mène vers une situation gagnante, a contrario du coup représenté par la branche de gauche qui l'amène vers une situation d'égalité. Le joueur adverse qui a deux coups possibles devrait choisir une position qui met en situation de défaite l'ordinateur mais étant donné le choix précédant d'aller vers la branche de droite, l'adversaire peut amener la partie uniquement vers des situations gagnantes pour l'ordinateur. Grâce à MinMax, ce dernier a trouvé le coup qui l'a mené à la victoire.

Cependant, on remarque que l'exécution de cet algorithme met beaucoup de temps si on l'implémente sur un ordinateur. Effectivement, si on cherche dès le début de la partie toutes les situations de jeux possibles pour tous les coups possibles, le nombre de coups devient vite très grand même pour un ordinateur (ce n'est pas vrai pour des jeux très limités tel que le morpion, mais c'est le cas pour Othello). On doit donc limiter le nombres de coups simulés dans MinMax.

2.2 L'algorithme MinMax limité

Cet algorithme est MinMax limité à une certaine profondeur dans l'arbre. Dans l'image précédente, on voit explicitement que si chaque situation de jeu implique approximativement deux coups jouables, le calcul de tout les coups jusqu'à la fin de la partie est impossible dans un temps raisonnable. Ainsi, l'algorithme va appliquer MinMax mais au lieu de s'arrêter lorsqu'il a atteint la fin de la partie et retourner la situation de fin du jeu, il va s'arrêter lorsqu'on a atteint la profondeur donnée et retourner l'évaluation de la situation de fin (si on est plus ou moins proche d'une situation de victoire).

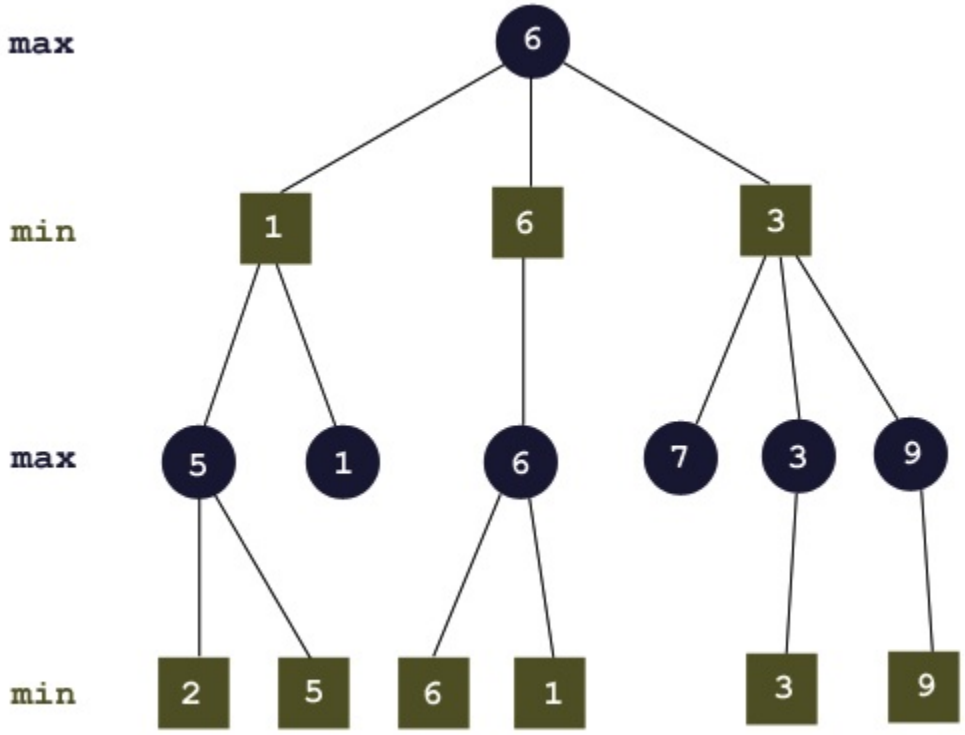

FIGURE 2.2 – MinMax limité par une profondeur donnée

Les feuilles de ce nouvel arbre sont les différents retours de la fonction d'évaluation pour les différentes situations de jeu. Pour les noeuds qui ne sont pas des feuilles, l'algorithme est similaire au MinMax vu précédemment : on regarde les valeurs des noeuds fils et on cherche alternativement à maximiser nos gains tout en minimisant les gains du joueur adverse. Voici un algorithme implémentant MinMax :

```
int fonction minimax (int profondeur){
        if (game_over or profondeur = 0)
                return evaluation();
        int meilleur_score;
        move meilleur_coup;
        if (noeud == MAX) { //=Programme
                meilleur_score = -INFINITY;
                for (chaque coup possible m) {
                        jouer le coup m
                        int score = minimax (profondeur - 1)
                        annuler le coup m;
                        if (score > meilleur_score) {
                                meilleur_score = score;
                                meilleur_coup = m ;
                        }
                }
        }
        else { //type MIN = adversaire
                meilleur_coup = +INFINITY;
                for (chaque coup possible m) {
                        jouer le coup m;
                        int score = minimax (depth - 1)
                        annuler le coup m;
                        if (score < meilleur_score) {
                                meilleur_score = score;
                                meilleur_coup = m ;
                        }
                }
        }
        return meilleur_coup ;
}
```

On peut maintenant imaginer la suite des evenements selon nos coups jouables mais dans une certaine mesure. Effectivement, de la profondeur dépendra la pertinence du choix d'un coup. Sachant que pour Othello le nombre de coups potentiels est de 8, un arbre de profondeur d aurait jusqu'à 8^d feuilles, donc on ferait 8^d appels à la fonction d'évaluation. Ainsi, si l'on veut calculer tous les coups jusqu'en profondeur 8 avec une fonction d'évaluation qui met 10^{-4} secondes à s'éxecuter, MinMax mettra $8^8 * 10^{-4} = 1678$ secondes, soit 28 min simplement pour calculer un coup. Il existe une optimisation permettant d'optimiser l'algorithme : l'élagage AlphaBeta.

2.3 L'élagage AlphaBeta

Lors de l'execution de MinMax, nous remarquons que certaines branches n'avaient pas besoin d'être développées pour ramener l'information des feuilles vers la racine.

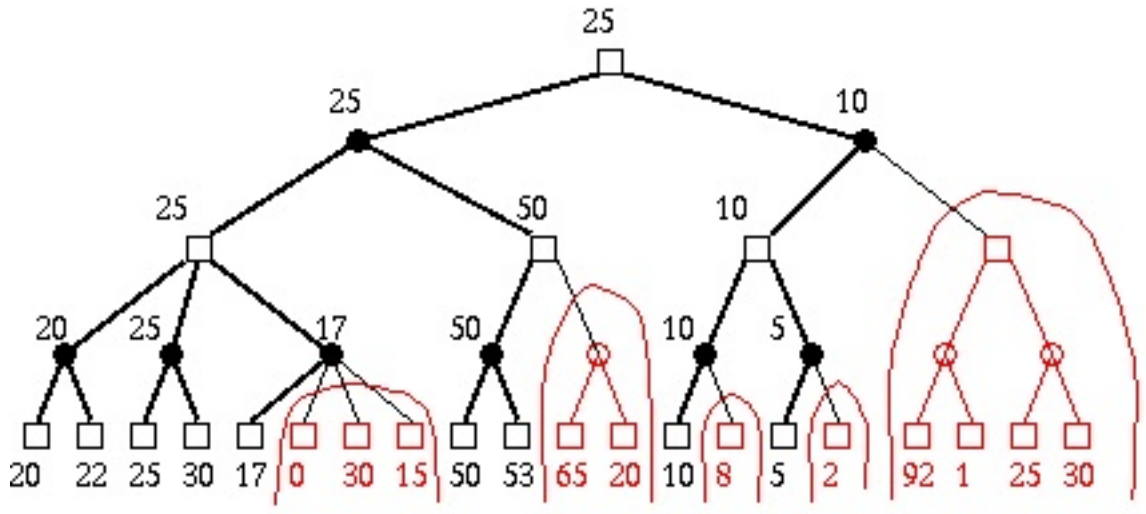

FIGURE 2.3 – L'élagage AlphaBeta (les branches rouges ne sont pas développées par AlphaBeta)

Sur cette image, le premier élagage permet de développer trois noeuds (et donc d'appeler la fonction d'évaluation trois fois de moins). On lit la valeur de la feuille 17 et on sait que l'on veut minimiser les gains, donc la valeur du parent sera inférieure ou égale à 17. Mais on voit que ce parent à des frères de valeurs supérieures à 17 alors on sait que ça ne sert à rien de chercher la valeur des frères de la feuille 17. En pratique, l'algorithme AlphaBeta a besoin de deux variables α et β, où α contient la valeur minimale que le joueur peut espérer obtenir pour le coup à jouer étant donné la position où il se trouve et β est la valeur maximale. AlphaBeta peut être implémenté comme ceci :

```
int alphabeta(int profondeur, int alpha, int beta)
{
        if (game_over or profondeur <= 0)
                return eval();
        move meilleur_coup;
        for (chaque coup possible m) {
                jouer le coup m;
                int score = -alphabeta(profondeur - 1, -beta, -alpha)
                annuler le coup m;
                if (score >= alpha){
                        alpha = score ;
                        meilleur_coup = m ;
                        if (alpha >= beta)
                                break;
                }
        }
        return alpha;
}
```

AlphaBeta permettra d'aller un peu plus loin en profondeur que MinMax. Cependant, depuis le début de ce chapitre, nous parlons d'une fonction d'évaluation qui serait capable d'évaluer une situation du jeu. Toute l'efficacité de l'algorithme MinMax dépend de cette fonction qui détermine les valeurs sur les feuilles de l'arbre.

Deuxième partie

La fonction d'évaluation

Chapitre

3 Les différents critères

Comme vu précédemment, la fonction MinMax permet de simuler des coups dans le but d'en déduire le meilleur coup jouable. Cependant, comment noter chaque coup que l'on simule de façon à différencier les bons coups des mauvais ? C'est pour répondre à cette problématique qu'intervient la fonction d'évaluation. Cette fonction E évalue un coup et son influence (positive ou négative) sur la partie à partir de plusieurs critères C_1, ..., C_n, d'importances repectives λ_1, ..., λ_n. Ainsi, l'évaluation se calcule comme suit :

$$E(partie) = \sum_{i=1}^{n} \lambda_i C_i(partie)$$

Pour l'évaluation, nous avons retenu certains critères : positionnement, coins, mobilité, parité et matériel.

3.1 Le critère de positionnement

Ce critère consiste à repérer quels sont les pions définitifs. Un pion est dit définitif quand il ne peut plus changer de couleur jusqu'à la fin de la partie. Par exemple un coin est un pion définitif puisqu'il ne peut pas être pris entre deux pions de la couleur adverse.

Il ne faut donc pas donner inconsidérément des coins à votre adversaire. Le moyen le plus simple d'éviter de perdre un coin est de ne pas jouer sur les cases adjacentes au coin, que l'on appelle cases X pour celles qui sont sur les diagonales a1-h8 et a8-h1 et cases C pour celles qui sont sur les bords.

Attention : Les bords ne sont pas des cases définitives et ne sont pas toujours bons à prendre, on peut souvent exploiter les faiblesses des bords adverses. Par exemple, dans le cas suivant :

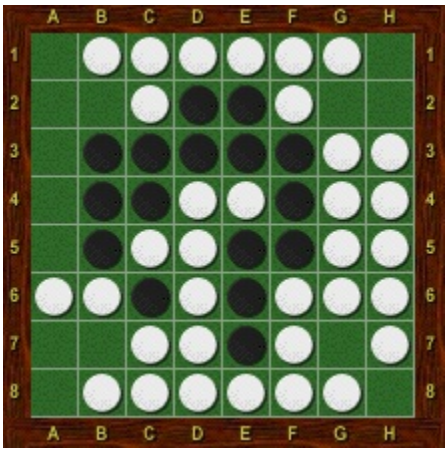

Figure 3.1 – un mauvais positionnement des bords

Blanc a beaucoup de bords et surtout, il a un "bord de cinq" sur la colonne h. Noir peut exploiter cette faiblesse en jouant la case g2. Certes, il perd alors d'emblée un coin si Blanc se précipite pour prendre h1, mais Noir peut alors s'insérer en h2 et gagner h8 au coup suivant (puis le coin a8). Blanc aura donc assuré 7 pions définitifs sur le bord nord, mais Noir en aura beaucoup plus à l'est comme au sud et gagnera la partie.

Il est donc important pour le programme de repérer et de prendre en compte les cases dites définitives dans les calculs MinMax.

3.2 Le critère des coins

Ce critère consiste à avantager les coups situés dans les quatre coins, aux extremités du damier. En plus d'être un pion définif, un pion situé dans un coin permettra la plupart du temps de retourner beaucoup de pions en fin de partie. De plus, il permettra de retourner des pions autour de ce coin, créant ainsi d'autres pions définitifs. Si l'on cherche à évaluer la partie après un coup de Noir, le calcul des coins sera le nombre de coins que possède Noir moins celui de Blanc. Un algorithme optimal qui implémente ce critère peut être présenté comme suit :

```
Fonction calculCoins(Partie : partie) : entier
    nombreCoins = 0, i : entier
    positionCoins[4] : entier [Tableau contenant la position des quatre coins sur le plateau de jeu]


    Pour i de 0 à nombreCoins-1 faire

        Si (plateau[positionCoins[i]] == couleurJoueur) Alors
            nombreCoins ← nombreCoins+1 ;
        Sinon
            Si (plateau[positionCoins[i]] == couleurAdversaire) Alors
                nombreCoins ← nombreCoins-1 ;
            Fin Si

        Fin Si

    Fin Pour
    Retourner nombreCoins ;
Fin
```

Algorithme 1: Calcul des coins

3.3 Le critère de mobilité

Ce critère consiste à forcer l'adversaire à jouer un mauvais coup (par exemple à donner un coin sans compensation). Pour ce faire il est important de posséder le moins de pions possible dans l'objectif de ne donner qu'un petit nombre de coups possibles à l'adversaire. En effet moins l'adversaire a de coups possibles, moins il a de possibilité de jouer un bon coup.

En général, plus vous placez de pions en frontière de la position, c'est-à-dire adjacents à des cases vides, plus votre adversaire a de coups et moins vous en avez.

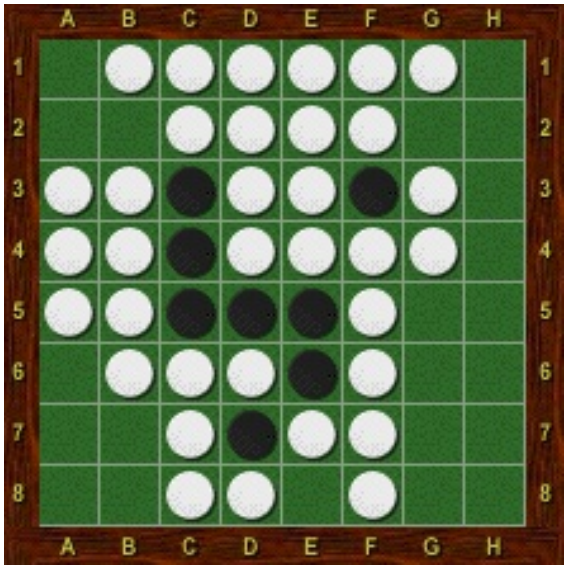

FIGURE 3.2 – un exemple de faible mobilité pour Blanc

Ici la mobilité du joueur blanc est très faible, en effet il n'a que deux coups possibles : b2, qui donne à Noir le coin a1 ou g2, qui donne le coin h1. De plus, comme Blanc a six pions sur le bord nord, dès que Noir prendra l'un des coins a1 ou h1, il pourra prendre l'autre au coup suivant. Noir a donc une très bonne mobilité.

Ce critère peut s'implémenter avec le pseudo-code qui suit :

```
Fonction calculMobilite(Partie : partie) : entier
    mobiliteJoueur = 0, mobiliteAdversaire = 0, i : entier

    Pour i de 0 à tailleTableau faire

        Si (plateau[i] est vide) Alors
            mobiliteJoueur ← mobiliteJoueur+1 ;
        Sinon
            changeCouleur() ; [On passe le tour]
            Si (plateau[i] est jouable) Alors
                mobiliteAdversaire ← mobiliteAdversaire+1 ;
            Fin Si
            changeCouleur() ;

        Fin Si

    Fin Pour
    Retourner mobiliteJoueur - mobiliteAdversaire ;
Fin
```

Algorithme 2: Calcul de la mobilité

3.4 Le critère de parité

Si aucun joueur ne passe son tour durant la partie, le joueur blanc joue le dernier coup de la partie et possède donc un avantage, en effet les pions retournés au dernier coup sont forcément définitifs. Le joueur noir a donc intérêt à essayer de faire passer le joueur blanc car le dernier coup est très important. Nous avons donc implémenté un algorithme qui renvoie **0** si le joueur qui vient de placer le coup placera le dernier pion (sauf passage de tour), et **1** sinon.

```
Fonction calculParite(nombreCoupsJoues : entier) : entier
    | Retourner nombreCoupsJoues % 2 ;
Fin
```

Algorithme 3: Calcul de la parité

3.5 Le critère du matériel

Enfin, ce critère est le critère le plus évident pour l'évaluation de la partie. Il ne tient compte que des scores des joueurs. Si l'on cherche à évaluer la partie après un coup de Noir, le calcul du matériel sera la différence entre le score de Noir et le score de Blanc. Ce qui ce traduit par :

```
Fonction calculMateriel(score[2] : entier) : entier
    | Retourner score[couleurJoueur] - score[couleurAdversaire] ;
Fin
```

Algorithme 4: Calcul du matériel

La fonction d'évaluation

Après avoir expliciter les critères pour le calcul de la fontion d'évaluation, nous pouvons avoir une idée de la forme que prendra cette fonction :

```
Fonction evaluer(Partie : partie*) : entier
    | Retourner λ1calculPositionnement(Partie)
    | + λ2calculCoins(Partie)
    | + λ3calculMobilite(Partie)
    | + λ4calculParite(Partie.nombreCoupsJoues)
    | + λ5calculMateriel(Partie.score) ;
Fin
```

Algorithme 5: Fonction d'évaluation

Cependant, il reste un problème de taille à régler. Après avoir défini ces critères, comment déterminer les coefficients ?

Chapitre 4

Les algorithmes génétiques

4.1 Introduction

Dans la partie précédente, il est expliqué en quoi la fonction d'évaluation est cruciale pour la réalisation d'une intelligence artificielle capable de rivaliser avec un humain dans une partie d'Othello. De plus, il a été établit qu'elle prend la forme d'une combinaison linéaire de critères. Néanmoins, la question du choix des cœfficients appliqués à ces critères reste toujours d'actualité.

On peut tout d'abord penser à chercher ces valeurs manuellement, mais malheureusement cela s'avèrerait très fastidieux et peu efficace, à moins d'avoir beaucoup de chance. Une autre solution consiste à se tourner vers les algorithmes génétiques, qui ont fait leurs preuves dans de nombreux domaines.

4.2 Présentation

Les algorithmes génétiques sont des algorithmes d'optimisation stochastique basés sur les principes fondamentaux de la sélection génétique naturelle.

De façon générale, on part tout d'abord un ensemble de solution potentielles, choisies aléatoirement parmi des valeurs que l'on estime plus ou moins intéressantes. Cet ensemble est appelé *population* et chaque élément de cet ensemble est appelé *individu*. Ensuite, on calcule leur efficacité par rapport au problème posé (leur *fitness* par rapport à l'*environnement*). On en déduit la nouvelle génération de cette population. Pour cela, on peut procéder à une *sélection* des meilleurs individus, à des *croisements* entre ces individus et enfin à des *mutations* sur certains gènes (avec une probabilité faible). On produit une nouvelle génération jusqu'à ce que le résultat obtenu soit satisfaisant.

De part la simplicité de leurs mécanismes, la facilité de leur mise en place et leur efficacité sur des problèmes complexes, les algorithmes génétiques sont de plus en plus utilisés dans des travaux mathématiques et économiques.

Un algorithme génétique est donc défini par des paramètres, qui concernent les individus, la population, l'environnement et le fitness, et par des opérateurs de sélection, de croisement et de mutation. La section suivante a pour but de définir ces critères.

4.2.1 Les paramètres

Ces paramètres définissent le cadre de l'application de l'algorithme.

Les individus

Un individu est représenté par une série de valeurs binaires, entières ou réelles selon le problème à traiter. Chacune de ces valeurs peut être assimilée à un gène. Chaque individu est une solution

potentielle au problème posé. Les valeurs de ses gènes sont tirées aléatoirement, mais de manière ciblée, lors de la création de la population.

La population

Une population est un ensemble d'individus de taille fixée au début de l'expérience. Elle engendrera d'autres populations de taille identique grâce aux opérateurs décrits plus bas.

L'environnement

L'environnement est l'espace dans lequel est situé le problème, celui dans lequel les individus évoluent. Il doit être définit dès le début de de l'algorithme.

Le fitness

Le fitness est une valeur donnée par la fonction $f(I)$, où f est la fonction de fitness et I un individu. Cette valeur positive représente le capacité d'intégration de l'individu dans l'environnement : plus elle est élevée, plus l'individu correspond à la solution du problème et plus il a de chance d'être sélectionné pour la génération suivante. La fonction f est donc déterminante pour le bon fonctionnement de l'algorithme.

4.2.2 Les opérateurs

Les opérateurs permettent, une fois que le fitness de chaque individu est calculé, la création d'une nouvelle génération. S'ils sont bien utilisés, les individus de la cette génération seront plus adaptés à l'environnement (i.e leurs fitness seront plus élevés).

La sélection

Cet opérateur permet de choisir les individus de la nouvelle génération en fonction de l'ancienne génération. Il ne modifie en rien les gènes des individus, mais peut en supprimer ou en dupliquer certains.

La méthode de sélection la plus utilisée (et celle qui semble la plus naturelle) est la *roue biaisée*. Pour chaque individu de la nouvelle génération, on le choisit au hasard parmi les individus de l'ancienne génération, de façon à favoriser les individus dont le fitness est le plus élevé. La probabilté $p(I)$ de choisir un individu I dans une population de taille N est donnée par la formule :

$$p(I) = \frac{f(I)}{\sum_{i=1}^{N} f(I_i)}$$

Ainsi, la probabilité de choisir un individu est proportionnelle à son fitness.

Néanmoins, cette méthode de sélection a quelques inconvénients : il est toujours possible qu'un individu ayant un bon fitness disparaisse de la population et donc qu'un individu ayant un bon fitness (mais pas le meilleur) domine la sélection.

Il existe plusieurs autres solutions pour remédier à ce problème. Il est possible de choisir automatiquement les n meilleurs individus pour la nouvelle génération et de sélectionner les autres par roue biaisée. On peut aussi sélectionner les individus par la méthode du *tournoi*, c'est-à-dire que l'on choisit aléatoirement deux individus dans la population et que l'on sélectionne celui qui a le meilleur fitness pour la nouvelle génération.

Les croisements

Même si l'opérateur de sélection permet d'améliorer le fitness total d'une génération, il ne permet pas de créer de nouveaux individus. C'est le rôle de l'opérateur de croisement.

Cet opérateur s'applique sur deux individus. Il faut tout d'abord déterminer s'il y a croisement ou non (cette probabilité p_c doit être fixée au début de l'expérience). Dans le cas d'un croisement, on tire un nombre aléatoire entre 1 et $l-1$ (on verra plus loin pourquoi il est inutile de tirer 0 ou l). Ce nombre est le *site de croisement*. Il représente l'endroit à partir duquel les gènes seront échangés chez les deux individus.

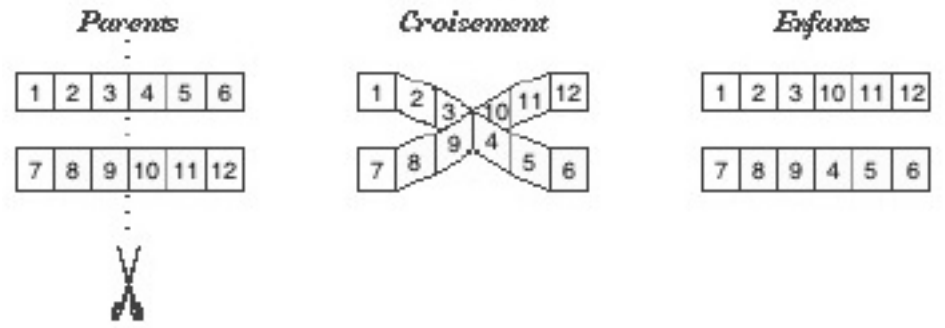

FIGURE 4.1 – croisement avec un site de croisement en position 3

L'illustration permet de se rendre compte qu'un site de croisement en 0 ou l aboutirait à l'échange pur et simple de tous les gènes des deux individus, et serait donc inutile.

Les mutations

L'opérateur de mutation est définit par la probabilté p_m de "muter" un gène, c'est-à-dire de lui attribuer une valeur aléatoire. Cette probabilité est généralement très faible est n'impacte pas beaucoup les résultats obtenus, mais elle permet de s'assurer que tous les points de l'environnement peuvent être atteints. En particulier, elle peut permettre de trouver un individu ayant un fitness supérieur à tous les autres et d'arriver à une meilleur solution finale. il est dont lui aussi très important.

4.2.3 Le déroulement de l'algorithme

Une fois que tous les critères et les paramètres des algorithmes génétiques sont définis, on peut établir le pseudo-code permettant une implémentation dans différents langages de programmation. Celui-ci est très simple, la difficulté résidant dans le fait d'implémenter les fonctions permettant la sélection, le calcul du fitness, le croisement, et les mutations.

```
Fonction generetation_suivante(ancienne_pop : population) : population
    nouvelle_pop : population

    [On crée une nouvelle population de la même taille que l'ancienne]

    nouvelle_pop = creer_population(ancienne_pop.taille)

    [On calcule le fitness]

    calculerFitness(ancienne_pop)

    [On sélectionne les nouveaux individus]

    selectionnerIndividus(ancienne_pop, nouvelle_pop)

    [On applique le critère de croisement]

    croiserIndividus(nouvelle_pop)

    [On applique le critère de mutation]

    muterIndividus(nouvelle_pop)

    [On détruit l'ancienne population]

    detruirePopulation(ancienne_pop)

    Retourner nouvelle_pop
Fin


Fonction fonction_principale() : None
    i : entier
    pop : population

    Pour i de 1 à nombreGénérations faire
        pop = generetation_suivante(pop)
    Fin Pour

    [Les individus de pop ont maintenant des gènes contenant les solutions au problème, ou des valeurs s'en
    approchant]
Fin
```

Algorithme 6: algorithme génétique standard

4.3 Les statistiques

Devant le nombre important de paramètres à régler, il nous a paru important de réaliser des études statistiques sur ceux-ci dans le but de savoir dans quelle mesure ils jouaient sur le résultat final de l'algorithme, mais aussi de connaître leurs valeurs optimales dans le cas particulier d'une fonction d'évaluation d'Othello.

Le calcul du fitness dans le cas du jeu d'Othello pouvant s'avérer long, nous avons choisi de

réaliser ces études en deux temps : tout d'abord une étude sur la forme, effectuée sur la *fonction de Rosenbrock* et permettant de découvrir la dépendance entre les différents critères, puis une étude rapide sur la fonction d'évaluation d'Othello, qui se basera sur les résultats de la première et d'après laquelle on pourra définir les valeurs optimales à fixer.

4.3.1 Les statistiques sur la fonction de Rosenbrock

Pour déterminer l'interdépendance des critères liés aux algorithmes génétiques, nous avons choisi d'utiliser le problème sur lequel nous avions travaillé en cours, à savoir la minimisation de la fonction de Rosenbrock. Il s'agit d'une fonction à deux variables décrite par la formule :

$$R(x, y) = (1 - x)^2 + 100(y - x^2)^2$$

Les valeurs de cette fonction étant toujours positives, l'environnement est $\Re^+$. Les individus sont définis par deux gènes, x et y, dont les valeurs seront initialement choisies dans l'intervalle $[0; 1]$. Les gènes d'un individu I sont notés respectivement I_x et I_y. Le fitness devant être positif et augmenter lorsque l'image par la fonction de Rosenbrock diminue (on rappelle que l'on souhaite minimiser cette fonction), on choisi la formule de calcul de fitness suivante :

$$f(I) = \mathrm{e}^{-R(I_x, I_y)}$$

Les valeurs du fitness seront alors toutes dans l'intervalle $[0; 1]$. Le fitness atteint 1 lorsque $R(I_x, I_y)$ vaut 0 et tend vers 0 lorsque $R(I_x, I_y)$ prend de grandes valeurs. Cette formule correspond donc bien au problème posé.

En partant sur ces bases, nous avons observé séparément les variations du nombre d'individus de la population, du nombre de générations calculées avant l'arrêt de l'algorithme, du rang de la population finale obtenue, du taux de mutation et du nombre d'individus étant sélectionnés avant d'utiliser la roue biaisée.

Le *rang* de la population finale est défini de manière récursive. Une population "classique" est une population de rang 1. Pour obtenir une population de rang $n + 1$, on sélectionne l'individu ayant le meilleur fitness dans N populations de rang n, avec N le nombre d'individus dans une population.

Les individus sélectionnés avant d'appliquer la roue biaisée sont bien entendu ceux ayant le meilleur fitness.

Par défaut, les paramètres suivants sont utilisés : 150 individus par population, algorithme répété sur 1000 générations, population finale de rang 1, taux de mutation de 1%, 0 individu sélectionné avant la roue biaisée et 0 individu sélectionné aléatoirement (les deux derniers paramètres sont définis plus loin).

En outre, le critère de croisement choisi est un peu particulier. En effet, dans le cas du croisement de deux individus P et M, on tire au hasard un nombre λ dans l'intervalle $[0; 1]$. On calcule ensuite les gènes du fils F par les formules :

$$\begin{cases} I_x = \lambda P_x + (1 - \lambda) M_x \\ I_y = \lambda P_y + (1 - \lambda) M_y \end{cases}$$

On génère ainsi un nouvel individu dont les gènes sont compris entre ceux de P et ceux de M.

Variation du nombre d'individus

Nous avons tout d'abord fait varier le nombre d'individus de la population entre 10 et 200, par palier de 2. Pour chaque valeur étudiée, l'algorithme a été lancé 200 fois. Le fitness moyen est alors la moyenne des 200 fitness moyens calculés.

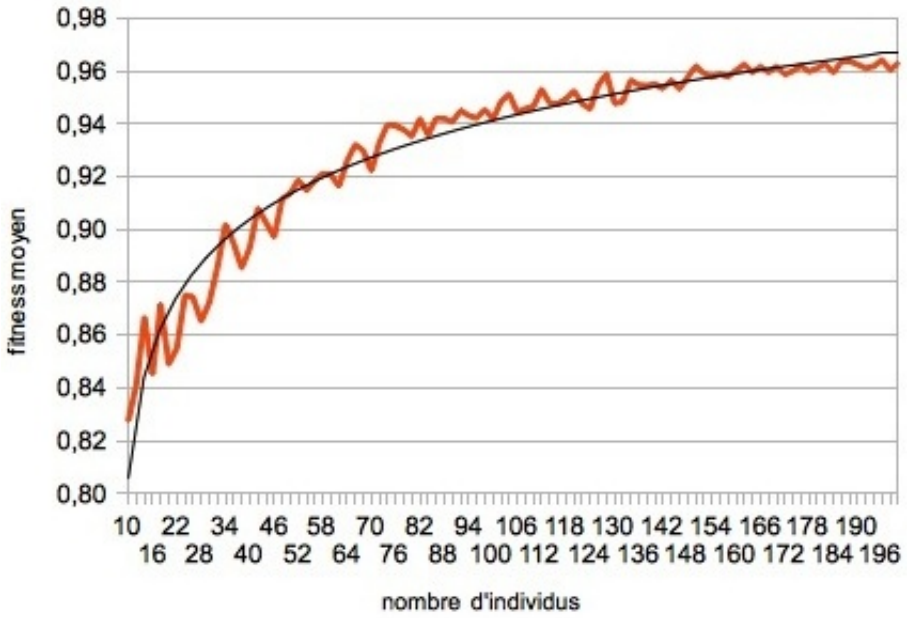

FIGURE 4.2 – Graphique de variation du fitness moyen en fonction du nombre d'individus dans la population

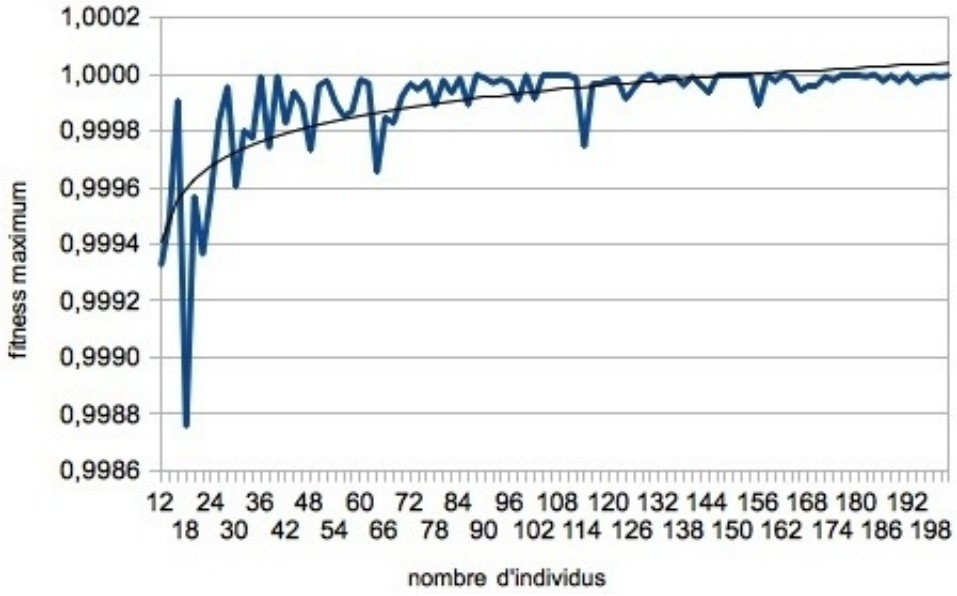

FIGURE 4.3 – Graphique de variation du fitness maximum en fonction du nombre d'individus dans la population

On remarque que la progression du fitness, que ce soit en moyenne ou en maximum, semble prendre une allure logarithmique : son augmentation diminue quand le nombre d'individus augmente. Il paraît donc judicieux de choisir une valeur assez grande pour avoir des résultats efficaces,

mais assez faible pour ne pas demander trop de temps de calcul. Par exemple, ici on pourrait choisir d'utiliser 160 individus par population, car la progression du fitness moyen passé cette valeur est négligeable par rapport au temps de calcul supplémentaire qu'elle demande.

Variation du nombre de générations avant maturité de la population

Nous avons ensuite étudié les effets des modifications du nombre de générations avant maturité de la population sur le fitness. Pour ce faire, nous l'avons fait varier de 100 à 2000 par palier de 10. Comme pour la variation du nombre d'individus, l'algorithme a été lancé 200 fois par valeur, afin d'avoir un résultat plus stable et donc plus exploitable.

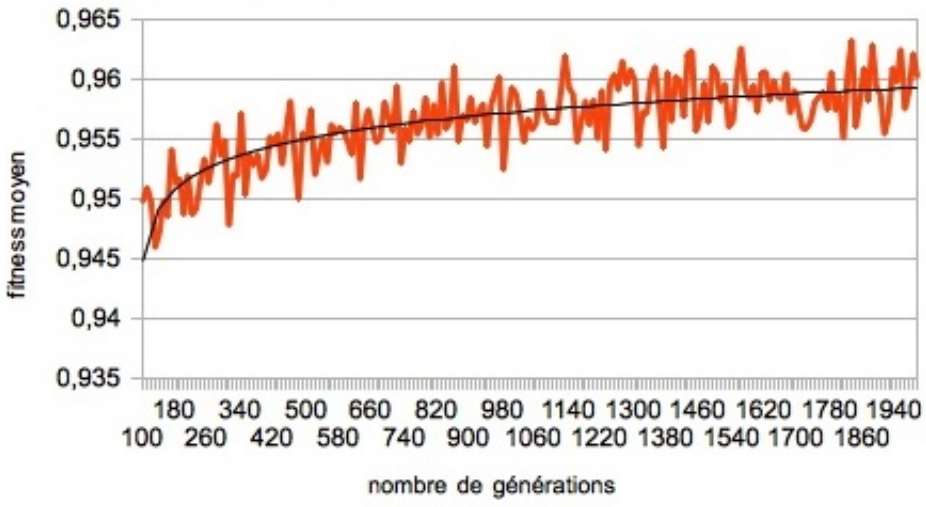

FIGURE 4.4 – Graphique de variation du fitness moyen en fonction du nombre de générations calculées

Comme pour le nombre d'individus, l'évolution du fitness moyen prend une forme logarithmique. Néanmoins, il est ici moins facile de déterminer une valeur à partir de laquelle l'amélioration du fitness est négligeable, car l'évolution semble quasi-linéaire à partir de 600 générations. Le choix de la valeur dépendra donc surtout du temps de calcul que l'on peut consacrer à une population (dans le cas de la fonction de Rosenbrock).

Le graphique montrant le fitness maximum n'est pas montré car celui-ci ne permettait pas de conclure.

Variation du rang de la population finale

Pour étudier le rôle du rang de la population finale, nous l'avons fait varier entre 1 et 3. L'augmentation de ce paramètre augmentant considérablement le temps de calcul, nous n'avons pas pu dépasser 3, et le nombre de lancements de l'algorithme n'a pas pu être constant au cours de l'exprience. Il a été de 2000 pour une population de rang 1, de 200 pour une population de rang 2 et de 50 pour une population de rang 3.

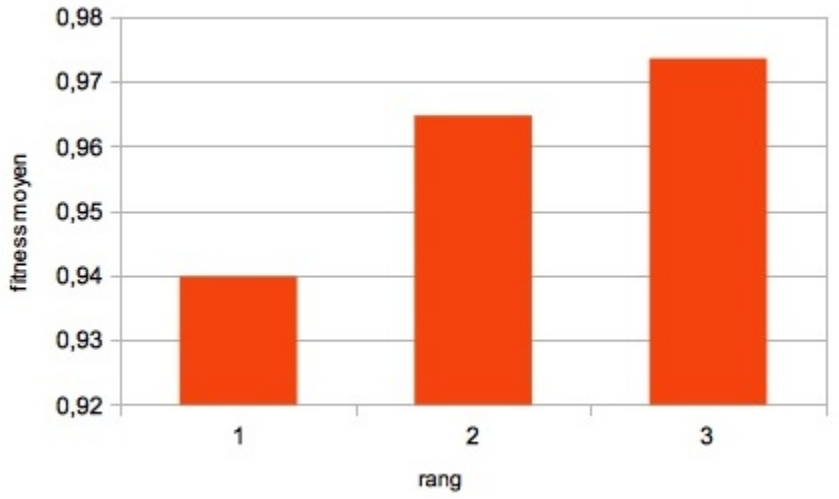

FIGURE 4.5 – Graphique de variation du fitness moyen en fonction du rang de la population obtenue

La graphique montre bien que plus le rang est élevé, plus le fitness moyen se rapproche de son maximum. Cela paraît logique car les individus d'une population de rang 2 sont déjà relativement adaptés à l'environnement, puisqu'ils sont issus d'une population vieille de 1000 générations.

L'augmentation du fitness moyen semble diminuer lorsque le rang augmente, mais il est difficile de conclure sur seulement 3 valeurs. Dans tous les cas, il sera difficile de dépasser le rang 3 pour une question de temps (mais ceci reste à voir dans le cas de l'étude d'une fonction d'évaluation d'Othello).

Variation du taux de croisement

Nous avons étudié la variation du taux de croisement. Pour cela, nous l'avons fait varier entre 0% et 100%, par palier de 1%. Chaque valeur est la moyenne de 200 fitness moyens obtenus.

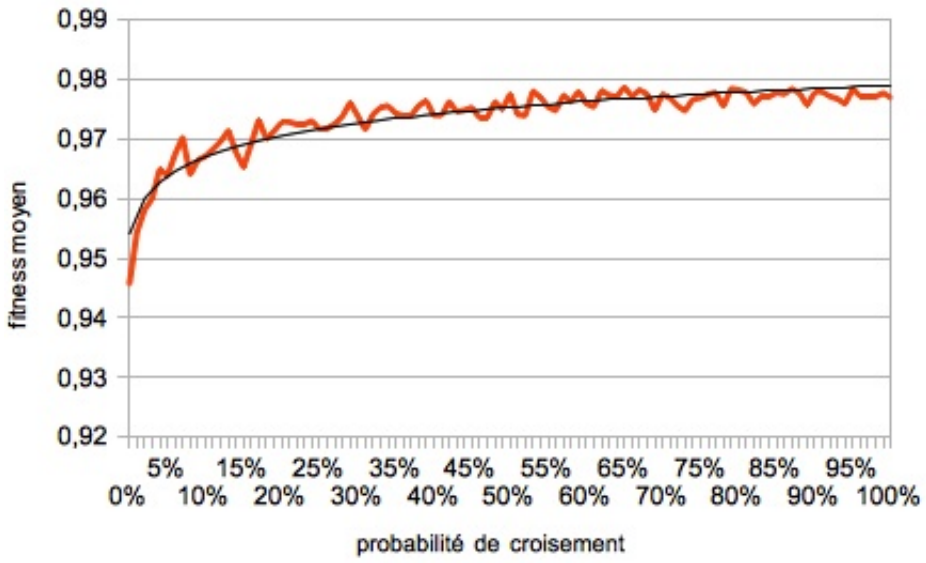

FIGURE 4.6 – Graphique de variation du fitness moyen en fonction du taux de croisement

On observe une progression logarithmique du fitness moyen en fonction de la probabilité de

croisement. Néanmoins, le croisement demande du temps de calcul et il sera donc judicieux de choisir une valeur du taux de croisement la plus basse possible, mais de manière à ce que le fitness moyen obtenu reste bon. Sur ce graphique, on pourrait choisir par exemple un taux de 50%, ce qui correspond à une probabilité de $\frac{1}{2}$.

Variation du taux de mutation

Le taux de mutation, quant à lui, à été étudié pour des valeurs comprises entre 0% et 2% (le pas entre deux valeurs consécutives étant de 0,01%). Pour chaque valeur, le fitness moyen retenu est la moyenne des 200 fitness moyens enregistrés.

Une première série a été lancée sur une population arrivée à maturité après 1000 générations. Les résultats n'étant pas très concluant, nous avons décidé de lancer une nouvelle série sur une population arrivée à maturité après 500 générations. Ainsi, l'allure de la courbe a pu être confirmée.

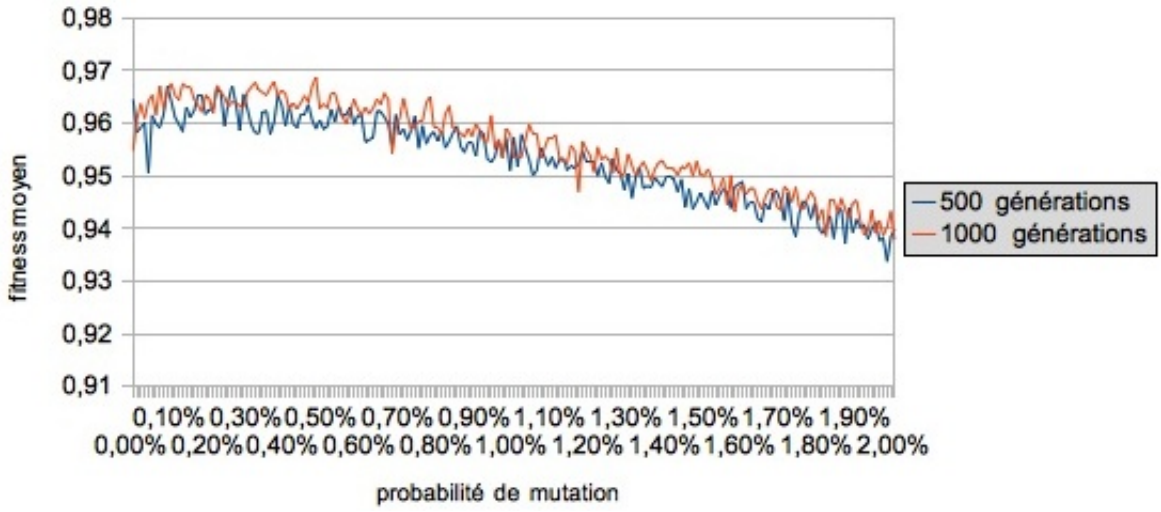

FIGURE 4.7 – Graphique de variation du fitness moyen en fonction du taux de mutation

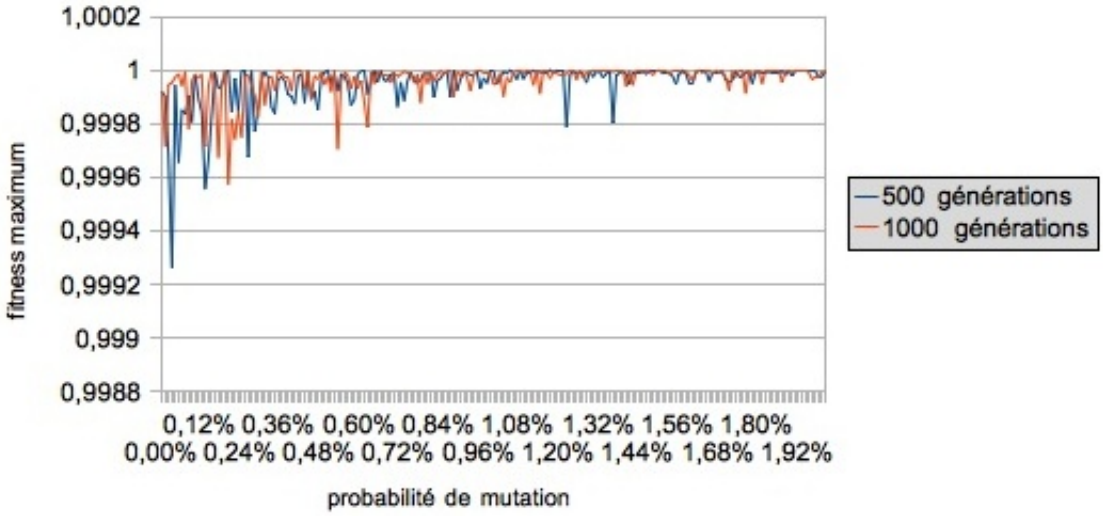

FIGURE 4.8 – Graphique de variation du fitness maximum en fonction du taux de mutation

On remarque sur la figure 6 que le fitness moyen semble être légèrement meilleur pour un taux compris entre 0,1% et 0,5%, puis qu'il chute jusqu'au taux de 2%, et ce sur les deux courbes. En outre, la figure 7 nous montre une amélioration du fitness maximum lorsque le taux parcourt les valeurs de 0% à 0,6%, puis se stabilise.

Choisir un bon taux de mutation revient dont à trouver un équilibre entre la maximisation du fitness moyen et celle du fitness maximum. Pour cela, il faudra donc choisir une valeur située à la limite supérieure de l'intervalle pour lequel le fitness moyen est le plus haut.

Variation du nombre des meilleurs individus sélectionnés directement

En outre, nous avons fait varier le nombre d'individus qui ne sont pas sélectionnés par roue biaisée entre 0 et 150 (la taille de la population), par palier de 1. Ces individus sont évidemennt ceux qui ont le meilleur fitness de la population, qui sont donc automatiquement sélectionnés pour survivre. Les fitness affichés ont été calculés sur 200 exécutions de l'algorithme par valeur.

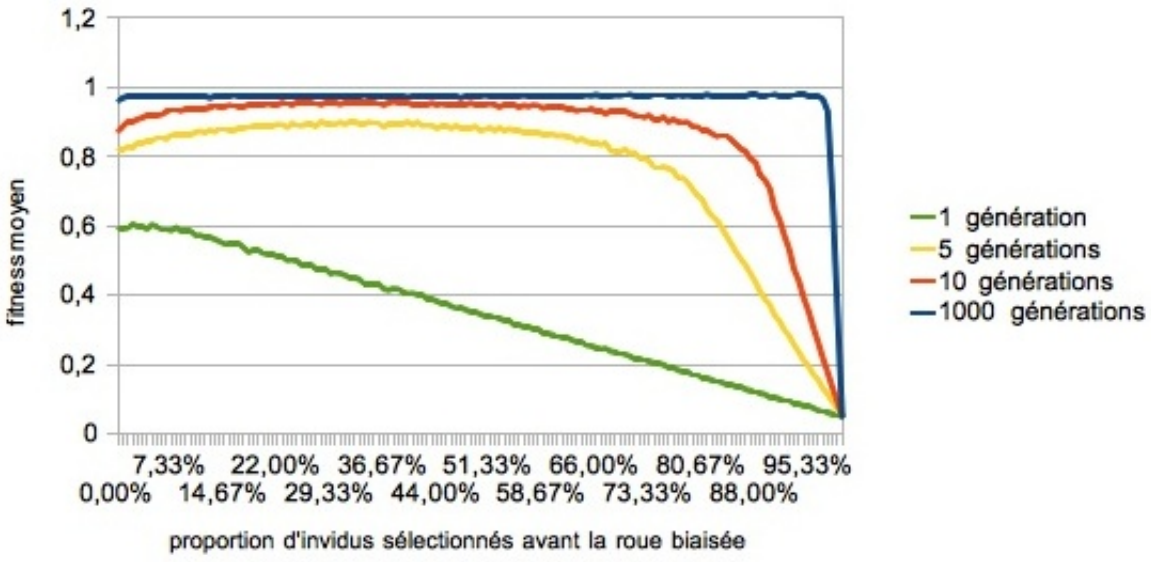

FIGURE 4.9 – Graphique de variation du fitness moyen selon le pourcentage d'individus sélectionnés pour leur fitness

Au vu des résultat obtenus lorsque l'algorithme calcule 1000 générations, nous avons été dans l'obligation de faire d'autres séries de statistiques. En effet, il semblerait que la quantité de générations annule les effets de la sélection automatique des meilleurs individus. Sur 1 génération, on observe une chute rapide du fitness moyen à partir d'environ 10% d'individus choisis pour leur fitness. Sur 5 et 10 générations, on s'aperçoit que cette méthode de sélection permet l'augmentation du fitness moyen par rapport à une sélection par roue biaisée, et que cette augmentation est optimale lorsqu'elle concerne environ 40% des individus.

Ainsi, cette méthode de sélection peut s'avérer efficace si le taux d'individus sélectionnés avant d'appliquer la roue biaisée est choisi de manière efficace et si le nombre de générations n'est ni trop élevé, ni trop faible.

Variation du nombre d'individus sélectionnées aléatoirement

Pour finir, nous avons décidé de choisir un certain nombre d'individus qui seront sélectionnés aléatoirement dans la population (chaque individu a la même probabilité d'être sélectionné, contrairement à la sélection par roue biaisée). Ceci permet de donner une chance aux individus

ayant un fitness faible de survivre. En effet, leurs génomes peuvent, croisés avec ceux d'autres individus, donner naissance à un individu dont le fitness est bon.

Nous avons donc fixé le nombre d'individus sélectionnés pour leur fitness à 25 et fait varier le nombre d'individus sélectionnés aléatoirement entre 0 et 125, par palier de 1. Les résultats sont exprimés selon le pourcentage de ces individus par rapport à la taille de la population. Comme précédemment, chaque valeur est la moyenne des 200 fitness moyens obtenus.

FIGURE 4.10 – Graphique de variation du fitness moyen selon le pourcentage d'individus sélectionnés au hasard

On constate que plus la proportion d'individus sélectionnés est importante, plus le fitness moyen est bas. Il semble donc obsolète de procéder à cette méthode de sélection.

Conclusion

Les résultats obtenus lors de cette étude statistique permettent d'avoir une approche *qualitative* des effets des paramètres sur les résultats de l'agorithme génétique. Ainsi, il est maintenant plus aisé de déterminer les valeurs précises à utiliser dans le cadre d'une fonction d'évaluation pour un jeu d'Othello. En effet, un faible nombre de valeurs sera suffisant pour déterminer la courbe associée et donc la valeur optimale.

4.3.2 Les statistiques sur les parties d'Othello

Il nous faut maintenant mener une étude quantitative afin de déterminer les paramètres que nous pourrons utiliser lors de la recherche des cœfficients de la fonction d'évaluation.

L'environnement choisi est une population d'individus dont les gènes sont tirés aléatoirement et qui servira de référentiel pour évaluer l'efficacité des autres individus. Les individus de cet environnement seront renouvelés régulièrement. Après un certain nombre de générations, on choisira quelques-uns des meilleurs individus de la population courante pour les y insérer. Ainsi, plus le nombre de générations augmentera, plus la population environnement aura un niveau de jeu élevé.

Les gènes d'un individu I sont une suite I_1, I_2, ..., I_n de flottants, qui correspondent aux coefficients appliqués à la fonction d'évaluation.

Pour calculer le fitness d'un individu, on le fait jouer contre chacun des individus de l'environnement. A chaque partie, on ajoute à son fitness la différence entre son score et celui de son adversaire.

Le critère de croisement est ici tout à fait standard.

Par défaut, les paramètres suivants sont utilisés :

- population calculée :
  - 50 individus
  - 100 générations
  - rang 1
- population environnement :
  - 50 individus
  - renouvelée toutes les 10 générations
  - renouvelée par lot de 10 individus
- opérateurs :
  - 1% de chance de mutation
  - 10% de chance de croisement
  - 5 individus sélectionné directement dans la population générée

Variation du nombre d'individus

Variation du nombre de générations avant maturité de la population

Variation du nombre d'individus dans l'environnement

Variation du nombre d'individus de l'environnement renouvelés

Variation du nombre du nombre de générations avant renouvellement de l'environnement

Variation du taux de mutation

Variation du taux de croisement

Variation du nombre des meilleurs individus sélectionnés directement