RODI

un programme d'Othello
de Nicolas Mazzocchi[1]

25/04/2013

[1] Étudient en deuxième année de licence informatique à la faculté de Marseille

Table des matières

I. Introduction au jeu d'Othello

a. Les règles
b. Stratégies de base
c. Historique

II. Approche algorithmique du jeu

a. Structures utilisées
b. Algorithmes principaux

III. Algorithme de recherche

a. Introduction
b. Algorithmes principaux
c. Tables de transpositions

IV. Fonction d'évaluation

a. Les patterns
b. La mobilité
c. Combinaison des critères
d. Découpage de l'évaluation

V. Apprentissage génétique

a. Structures utilisées
b. Algorithmes principaux
c. Convergence des échantillons

VI. Vue sur le programme

a. Aspect modulaire
b. Résultats observés
c. Mode d'emplois

Préambule

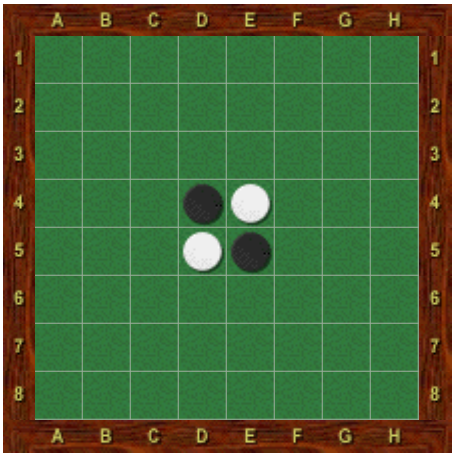
RODI est un projet algorithmique de deuxième année de licence informatique. Cet exercice présente la programmation modulaire et de la conception d'algorithmes de grande ampleur dans un cadre scolaire. L'impacte de ce projet sera d'une part, de développer les performances de programmation dans le langage C et consolider les connaissances déjà acquises. D'autre pars, la division du programme en modules adaptés à la compilation séparée est une approche du type 'diviser pour régner'. Cette méthode et l'un des fondements de l'informatique d'aujourd'hui. De plus, le jeu d'Othello permettra l'élaboration une intelligence artificiel utilisant l'algorithme récursif min-max avec élagage alpha-bêta et une fonction d'évaluation génétique.

I. Introduction au jeu d'Othello

a. Les règles

Othello, aussi connu sous le nom de Réversi, est un jeu de stratégie à deux joueurs. Il se joue sur un plateau de 64 cases appelé othellier. Le jeu disposent de 64 pions bicolores, noirs d'un côté et blancs de l'autre. Un pion est dit noir (respectivement blanc) si sa face visible est celle de couleur noire (respectivement blanc).

Un pion

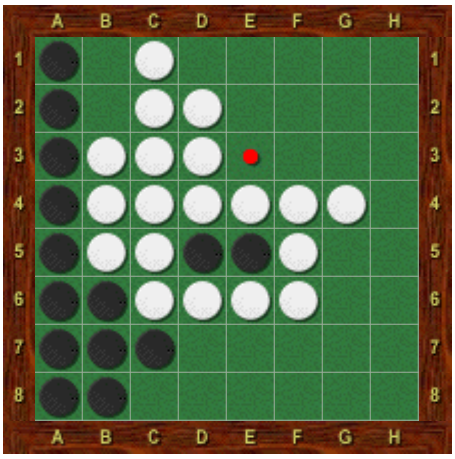


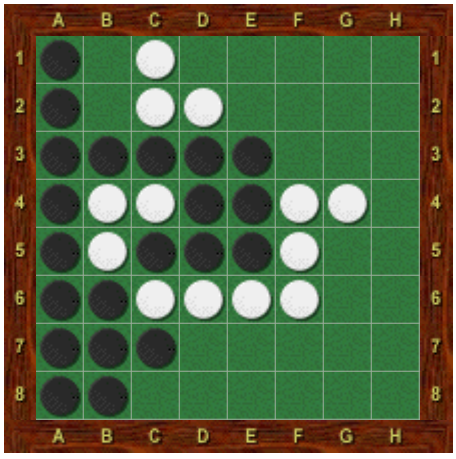
Le but du jeu est d'avoir plus de pions que son adversaire à la fin de la partie. Celle-ci se termine lorsque aucun des deux joueurs ne peut plus jouer de coup qui soit légal.

Au début de la partie, deux pions noirs sont placés en e4 et d5 et deux pions blancs sont placés en d4 et e5. Contrairement au dames ou au échecs, le joueur ayant les pions Noir commence toujours !

Le joueur qui a le trait doit poser un pion de sa couleur sur une case vide de l'othellier qui soit adjacente à un pion adverse. Pour que le coup soit légal, le pion posé doit encadrer un ou plusieurs pions adverses avec un autre pion allié déjà placé sur l'othellier. Pour chaque direction, il retourne les pions qu'il vient d'encadrer du côté de ça couleur. Si, le joueur ayant le trait ne peut pas poser de pion suivant les règles, il devra passer son tour par défaut. En revanche, s'il peut réaliser un coup qui soit légal, il ne pourra pas passer son tour.

Voici un exemple :



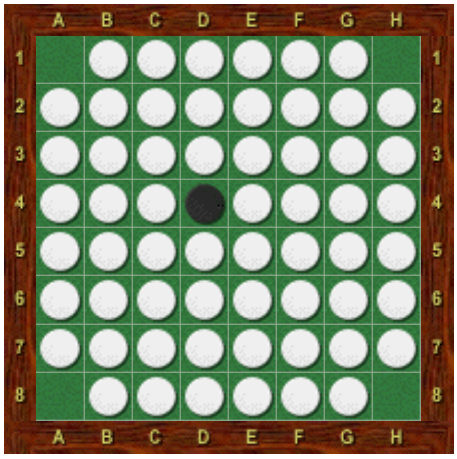


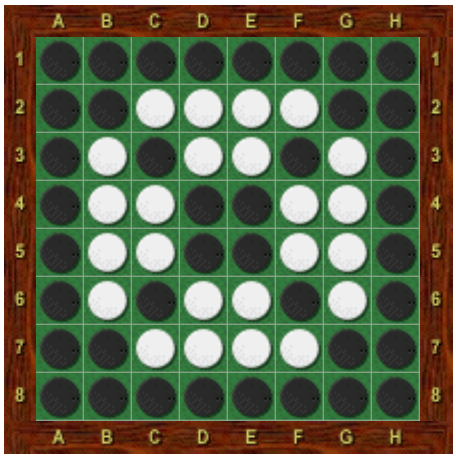
b. Stratégies de base

Pour un humain ce jeu est difficile à maitriser et par ailleurs la méthode qui semble la plus intuitive est en fait la moins efficace.

Maximisation :

Comme la configuration ci-dessous le démontre, la stratégie qui consiste à retourner à chaque coup le plus grand nombre de pions possible est la plus mauvaise. Dans la configuration suivante, Noir va jouer tous les coups restants (en commençant indifféremment par a1 ou h8, puisque Blanc passera chaque fois son tour) et l'emportera 40-24.

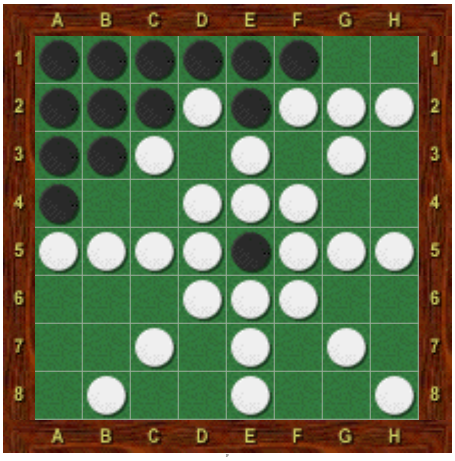




Avoir beaucoup de pions, même à ce stade de la partie, n'est donc pas suffisant pour assurer la victoire. Il est plus important de détenir des pions "définitifs" qui ne pourront plus être retournés.

Positionnement :

Il est impossible de prendre un coin en tenaille entre deux pions : Un pion placé dans un coin est donc l'exemple le plus simple de pion définitif. Les pions adjacents et de la même couleur deviennent aussi très souvent des pions définitifs. Cette constatation nous amène à la définition d'une position définitive : Une position qui ne possède pas de pion n'est pas définitive. Une position définitive le reste jusqu'à la fin de la partie. Une position qui possède quatre positions limitrophe successif définitif ou bien dont la direction n'admet plus de case vide devient définitive. On peut donc penser que c'est une bonne stratégie de prendre les bords en espérant que l'adversaire sera forcé de donner un coin tôt ou tard ce qui assurera plusieurs positions définitives.



Pions définitifs en noir

Mobilité :

Cette stratégie consiste à forcer l'adversaire à jouer un mauvais coup. Si l'adversaire a peu de coups possibles et si, de surcroît, ils sont mauvais, il sera bien obligé, en raison des règles, d'en jouer un. En revanche, s'il a le choix entre de nombreux coups, il en trouvera certainement un bon. Il faut donc minimiser le nombre de coups possibles de l'adversaire et maximiser ceux alliés.

Parité :

L'astuce de la parité est très simple à comprendre et à implémenter.

L'idée, est d'avoir le dernier coup. Par définition, le joueur qui joue en dernier, prends des pions à l'adversaire de manière définitive.

c. Historique

Écrit par Michael Buro en 1995, Logistello est un programme informatique du jeu Othello. Après avoir battu le champion du monde humain Takeshi Murakami six matchs contre zéro en 1997, Logistello est considéré comme étant le meilleur joueur d'Othello toutes catégorie confondu.

En effet, Othello est un jeu qui permet à l'ordinateur de devenir rapidement bien plus fort que l'humain. Il est donc assez simple de concevoir une intelligence artificiel qui puisse systématiquement battre un débutant d'Othello. En outre, Othello est un jeu simple. Contrairement au jeu d'échec, qui dispose de nombreuses pièces, ou le jeu de go est possède de multiple combinaison !

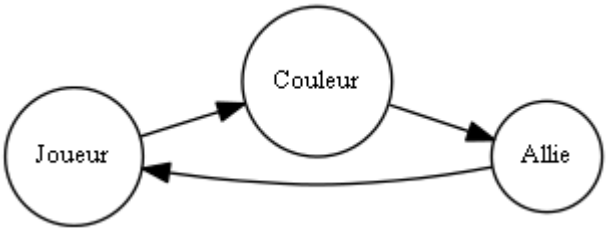
II. Approche algorithmique du jeu

a. Structures utilisées

Constantes :

BLANC = 1 et NOIR = 0

Ce choix de valeur pour les couleurs est extrêmement pratique car il correspond aux valeurs des indices des joueurs. Ainsi, dans le programme, l'indice d'un joueur est une couleur et une couleur est l'indice d'un joueur ! Par ailleurs dans l'algèbre de Boole 0 = **NON** 1.



Remarque, du fait de cet tri-équivalence, et par abus de langage, on utilisera le terme allier et adversaire pour désigner un joueur ou une couleur d'un pion en référence à une couleur représentative ou, un joueur représentatif (autrement dit dans un certain contexte d'une fonction).

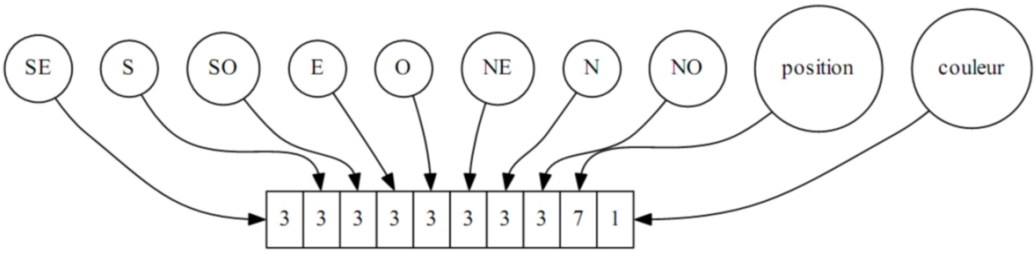
VIDE = 2 et BORD = -1

Ces deux nouvelles constantes concernent l'othellier à part entière. Une case peut être vide (VIDE) et si on va trop loin sur l'othellier on arrive sur un bord (BORD). La pertinence de cette remarque va naitre dans l'explication de la structure de l'othellier. De même, le choix de la valeur de VIDE sera expliqué dans la partie des patterns.

Un coup :

Un coup peut être représenté par le joueur qui joue ce coup, autrement dit la couleur du pion jouer puisque l'implémentation des couleurs et des joueurs est la même dans ce programme… Il est important d'y ajouter la position de la case jouée. De plus, d'un point de vue contextuel, un vecteur donnant de nombre de retournement pour chaque direction.

Les deux premier champs étant totalement évidant, notons plutôt l'idée de retournement. Cette façon d'implémenter la structure va permettre l'annulation d'un coup. En effet, pour un othellier, une position et une couleur donnés : il est impossible de reconstruire l'othellier tel qu'il été avant le dernier coup ! L'information du nombre de pions à retourner pour toutes les directions permet cette manipulation.



Du point de vu de la machine, un coup sera codé sur un entier de 32 bits. Le bit de poids le plus faible (à droite sur le schéma) code la couleur, les sept bits qui suivent codent la position et tous les bits restant code trois par trois les cases du vecteur de retournement.

Un othellier :

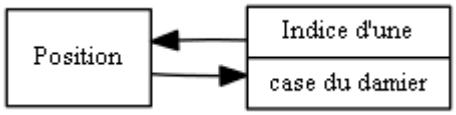
Part définition un othellier est un plateau de jeu de 64 cases (8x8). Pour s'affranchir de nombreuses contraintes d'existence d'indices, on choisi un vecteur de 100 cases. Dans ce cas de figure 36 cases seront donc présentatrice des bords de l'othellier. Cette représentation permet de considérer que deux nouvelles couleurs existes : le vide et le bord. Quand, par exemple, on ce déplace dans une certaine direction depuis une position, on devra s'arrêter si l'on est sur un bord : Un seul test sera fait et il ne dépendra que du contenue de la case. Cette représentation est donc très avantageuse, toutefois on ajoutera à la structure un petit vecteur de deux cases contenants chacune le nombre de pion de chaque joueur. Cet ajout permettra d'éviter de recompter tous les pions systématiquement.

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49
50	51	52	53	54	55	56	57	58	59
60	61	62	63	64	65	66	67	68	69
70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89
90	91	92	93	94	95	96	97	98	99

Damier d'un othellier

Sur le schéma ci-contre, on peut lire la correspondance entre les indices du vecteurs et les cases qu'ils représentes. Les bords du damier figures en marron, les case vide en vert et le case occupée de la couleur du pion.

Remarque, lorsque l'on parlera de position ce sera toujours en référence au damier et plus précisément à un indice du vecteur qui représente le damier.



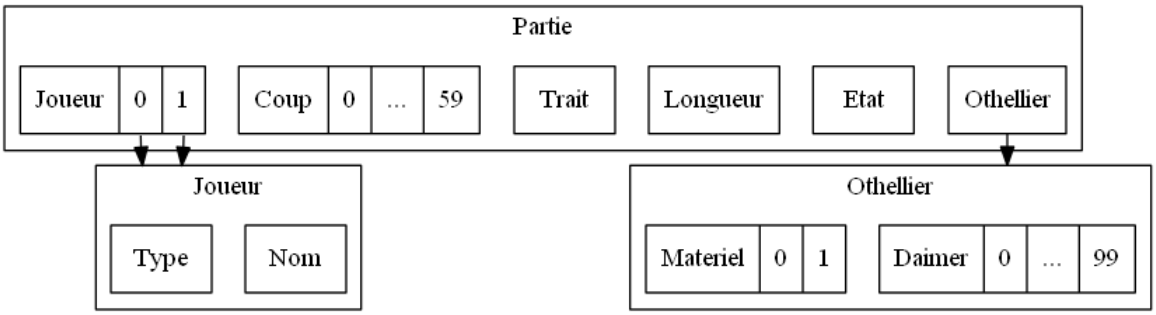
Un joueur :

Le joueur est une structure très simple. Comme dans la vie de tout les jours on attribut un nom à chaque personne. C'est ce qui va être fait pour cette structure. En outre, en petit plus doit intervenir du faire de l'intelligence artificielle : Le type. Le type d'un joueur est une énumération des niveaux de difficultés. Il est nul si le joueur est humain, autrement, il codera la difficulté de l'intelligence artificielle (de 1 à 3 car il a été choisi de faire trois niveau de difficulté). Remarque, l'état du code source actuel permet de réaliser neuf niveau.

Une partie :

La partie sera la structure la plus complexe du programme. Elle se composera de deux joueur, d'un othellier, et de 60 coup. Il serait judicieux d'y ajouter une valeur qui porterait la couleur du joueur qui doit jouer. Cette structure à l'air complète, toutefois, au cour de la programmation, il sera question de ce situer dans le vecteur des coups, en d'autres thermes, de connaître à quel coup on se situe. Pour ce faire, il

va falloir introduire les valeurs traditionnelle d'une boucle, soit, un incrément que l'on nommera état et une valeur maximale que l'on nommera longueur.



Récapitulatif des structures

b. Algorithmes principaux

Légalité d'un coup :

Cet algorithme permet de vérifier si un coup et légal ou pas. Par ailleurs, si le coup et légal les retournements seront actualisés. On va parcourir toutes les directions (NO, N, NE, O, E, SO, S, SE) et regarder si on n'est dans une configuration d'encadrement. En parallèle on comptera exactement combien de pion on encadre.

Entrées : coup (un coup), othellier (un othellier)
Sortie : **vrai** si le coup est légal **faux** sinon. Le coup est actualisé.
Variables : bool (un booléen), position (un entier), retournement (un entier)

```
début
{
  bool ← faux    # On suppose par défaut que le coup n'est pas légal

  si la position du coup ≠ VIDE alors
    retourner faux;
  # S'il y a déjà un pion sur la position demandée le coup est illégale

  pour toutes directions faire
  {
    position ← position du coup    # Position initiale
    position ← position suivante sur la direction    # Le case voisine

    retournement ← 0 # On suppose que aucun pion adverse n'est encadrés

    tant que position = NON couleur du coup faire
    # Tant qu'on est dans le cas où position est à coté de pions adverses
    {
      position ← position suivante sur la direction
      # On se déplace dans la direction

      retournement ← retournement + 1
      # On ajoute 1 au nombre de pion adverse supposé encadré
    }
    si position = couleur du coup ET retournement ≠ 0 alors
    # Si on est dans la configuration d'un encadrement
    {
      bool ← vrai # Le coup est légal
      retournement du coup pour la direction ← retournement
    }
```

```
    sinon
    # On peut être sur un bord ou avoir deux pion allié cote à cote
      retournement du coup pour la direction ← 0
  }
  retourner bool;
}
fin
```

Validation d'un coup :

Cet algorithme permet de poser un pion sur l'othellier et d'effectuer les retournements nécessaire. De plus, le matériel de l'othellier et actualiser. L'algorithme suit ce que tout bon joueur ferait lorsque qu'il pose un pion.

Entrées : coup (un coup), othellier (un othellier)
Sortie : othellier après le coup jouer
Variable : position (un entier), retournement (un entier)

```
début
{
  position du coup sur le damier ← couleur du coup (allié)
  nombre de pion allié ← nombre de pion allié + 1
  # On pose un pion sur la position du coup

  pour toutes directions faire
  {
    position ← position du coup
    position ← position suivante sur la direction
    # On regarde à coté de la position initiale

    retournement ← nombre de retournement dans la direction

    nombre de pion allié ← nombre de pion allié + retournement
    nombre de pion adverse ← nombre de pion adverse - retournement
    # On retourne retournement de pions adverse du coté allié

    tant que retournement n'est pas nul faire
    # On avance autant de fois que le nombre de pion que l'on encadre
    {
      on retourne le pion sur la position position du damier

      position ← position suivante sur la direction
      # On se déplace dans la direction

      retournement ← retournement - 1
    }
  }
}
fin
```

Annulation d'un coup :

Cet algorithme permet de revenir à l'état de l'othellier avant que le dernier coup soit jouer. L'algorithme fait l'opposé de la validation d'un coup.

Entrées : coup (un coup), othellier (un othellier)
Sortie : othellier à l'état qu'il avait avant le dernier coup jouer
Variable : position (un entier), retournement (un entier)

```
début
{
  la position du coup sur le damier ← VIDE
  nombre de pion allié ← nombre de pion allié - 1
  # On retire le pion posé

  pour toute direction faire
  {
    position ← position du coup
    position ← position suivante sur la direction
    # On regarde à coté de la position initiale

    retournement ← nombre de retournement dans la direction

    nombre de pion de allié ← nombre de pion allié - retournement
    nombre de pion adverse ← nombre de pion adverse + retournement
    # On retourne retournement de pions adverse du coté allié

    tant que retournement n'est pas nul faire
    # On avance autant de fois que le nombre de pion que l'on encadre
    {
      on retourne le pion sur la position position sur le damier

      position ← position suivante sur la direction
      # On se déplace dans la direction

      retournement ← retournement - 1
    }
  }
}
fin
```

Initialiser l'othellier :

Cet algorithme permet de configurer l'othellier pour commencer une partie. On commence par déterminer les BORDS, puis on place les pions initiaux et on affecte le matériel initial.

```
Entré : othellier (un othellier)
Sortie : othellier initialiser
début
{
   pour toutes position sur l'othellier faire
   {
     si position[10] = 0 OU position[10] = 9 \
     OU position/10 = 0 OU position/10 = 9 alors
     # si la position est ?0 ou 0? ou ?9 ou 9?
       position ← BORD
     sinon
       position ← VIDE
   }

   les positions d4 et e5 ← BLANC
   # Pions blanc initiaux
   les positions d5 et e4 ← NOIR
   # Pions noir initiaux
   nombre de pion BLANC ← 2
   nombre de pion NOIR ← 2
}
fin
```

III. Algorithme de recherche

a. Introduction

Comme la plus part des jeus de réflexion Othello est un jeu dits 'jeu à deux joueurs, à somme nulle et information complète'. A somme nulle car Les gains d'un joueur sont exactement l'opposé des gains de l'autre joueur (ce qui me fait gagner fait perdre mon adversaire). A information complète car lors du choix d'un coup à jouer chaque joueur connaît précisément ses possibilités d'action et celle de son opposant. De plus il connait les gains résultants de ces actions.

L'algorithme utilisé pour cette configuration est l'algorithme min-max auquel certaines améliorations pourront être apportées. Min-max est un algorithme qui reproduit la manière naturelle qu'un humain mettrai en œuvre pour jouer. Il consiste donc à anticiper la réplique de l'adversaire à un coup joué, puis la réplique de cette réplique … ainsi de suite jusqu'à une certaine condition d'arrêt. Une fois cette condition vérifiée, l'algorithme utilisera une fonction d'évaluation qui permettra de rendre compte de la situation dans laquelle se trouve un joueur étant donné l'othellier. Min-max permet de retourner l'évaluation correspondant au meilleur compromis maximisation-minimisation pouvant être fait à partir d'une situation. Soit la meilleur évaluation en supposant que l'adversaire ai une évaluation équivalente du jeu.

L'arbre généré par la récursion de min-max aura autant de branches que de case jouables sur le damier et ce pour tous les nœuds de cet arbre : La complexité de min-max est exponentiel. En effet, puisqu'en moyenne un joueur d'Othello à 8 choix de coup, alors, pour un arbre de hauteur h on observera 8^h fonction d'évaluation calculées. La complexité de l'algorithme nous contraint donc (la plus part du temps) à ne pas attendre la fin de partie pour restreindre la profondeur, mais plutôt à fixer une profondeur avant de commencer la recherche.

Min-max se décompose intuitivement en deux blocs : la maximisation de l'évaluation (pour l'allié) et sa minimisation (pour d'adverse). Or, du fait de l'opposition des gains et des pertes entres les deux joueurs, on pourra regrouper ces deux blocs en un seul au prix d'une simple inversion du signe du résultat renvoyé : On parlera plus de min-max mais de néga-max.

Attention, néga-max simplifie essentiellement l'esthétique de min-max, la recherche n'est pas plus rapide (car un test ne coute pas grand chose). Dans min-max comme pour néga-max, on remarque que le parcourt de l'arbre est exhaustif, ce qui n'est pas forcement une bonne chose pour un algorithme exponentiel. On peut donc accélérer la recherche en élaguant certaine branche de l'arbre (en éliminant certaine possibilité de jeu) qui ne serait pas utile au résultat retourné : C'est le principe de alpha-bêta.

L'élagage de alpha-bêta serra utilisé de tel sort à ce qu'il ne modifie pas le résultat que min-max aurait pu calculer (voir la preuve de l'alpha-bêta). Il serra néanmoins plus rapide.

b. Algorithmes principaux

L'algorithme alpha-bêta :

L'algorithme renvoie le meilleur score évalué pour une profondeur donné. En parallèle, la position qui permet de réaliser ce meilleur score est actualisé uniquement à la racine de l'arbre généré.

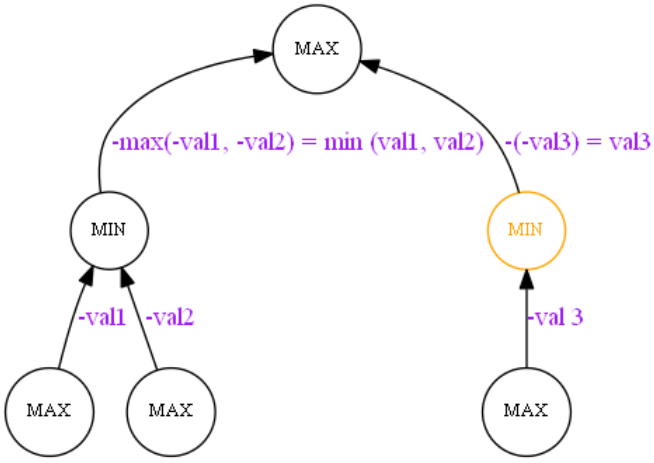
Gestion d'un tour passé :

Le principe néga-max ce base sur deux propriétés :

A = -(-A)

min(A, B) = -max(-A, -B)

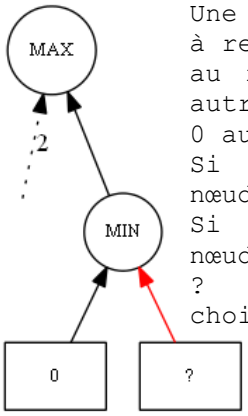
Ainsi, si à une hauteur h de l'arbre le joueur est le même que la hauteur h-1 alors l'algorithme réalisera à tort une inversion de signe ce qui faussera les évaluations. C'est en fait une fausse difficulté qui est très bien expliqué dans les commentaires de l'algorithme. Le schéma ci-contre sera un peut plus concret que du texte.



Dans l'algorithme, lorsque qu'un joueur passe son tour, la profondeur n'est pas décrémentée car le calcul est immédiat.

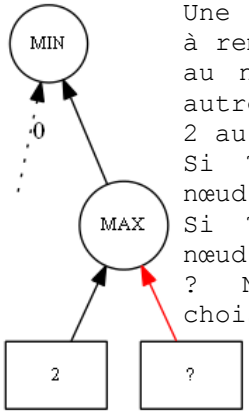
Principe fondamentale de l'élagage :

Une branche est élaguée, on dit aussi coupée, lorsque les valeurs des feuilles de cette branche n'a plus d'importance.



Une feuille inconnue à remonté la valeur 2 au nœud max. Et une autre feuille remonte 0 au nœud min.
Si ? ≥ 0 alors le nœud min choisira 0.
Si ? ≤ 0 alors le nœud max choisira 2.
? Ne sera jamais choisi !

Coupure alpha

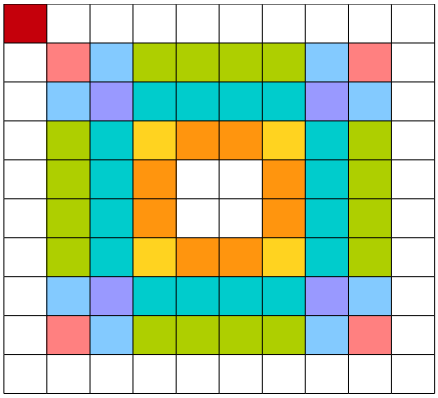


Une feuille inconnue à remonté la valeur 0 au nœud min. Et une autre feuille remonte 2 au nœud max.
Si ? ≥ 2 alors le nœud min choisira 0.
Si ? ≤ 2 alors le nœud max choisira 2.
? Ne sera jamais choisi !

Coupure bêta

Optimisation de l'élagage :

L'idée est de privilégier les positions puissantes pour optimiser le nombre de coupures alpha-bêta. Une matrice constante permet de réorganiser l'ordre de parcourt du damier. Ce parcourt optimal est visible en suivant les couleurs du spectre de la lumière sur le schéma ci-contre. Pour retrouver les valeurs de la matrice il suffit de regarder à quel indice du damier la case correspond.



Représentation du parcourt

Notons qu'il 61 valeurs pour 60 positions de jeu ! En fait, dans l'algorithme alpha-bêta on voudrait avoir la position suivant selon l'ordre optimal : Il faut donc une case conventionnelle d'initialisation, ici 0. En outre, les quatre cases centrales n'apparaissent pas dans la matrice pour la simple raison qu'elles sont déjà occupé par un pion du fait de l'initialisation.

Preuve de l'alpha-bêta :

On voudrai que l'alpha-bêta retourne le même résultat que min-max. Pour ce faire, on notera w = min-max (p) et v = alpha-bêta(p, α, β) avec toujours α < β.

Du fait des différents élagages, on observe que :
Si $v \leq \alpha \rightarrow w \leq v$
Si $\alpha < v < \beta \rightarrow w = v$
Si $v \geq \beta \rightarrow w \geq v$

Ces assertions sont assez simple à démontrer par récurrence.

BASE : Pour la profondeur p = 0

Quelque soit l'algorithme, c'est la fonction d'évaluation sera calculé.
w = évaluation = v

HEREDITE : Supposons les trois propriétés vérifiés en profondeur p-1

Si la partie est fini : w = évaluation = v

Si $v \leq \alpha$
→ -alpha-bêta (p-1, -β, -α) ≤ v
→ alpha-bêta (p-1, -β, -α) ≥ -v
→ min-max (p-1) ≤ v
→ -min-max (p-1) ≥ v
→ min-max(p) ≥ v
→ w ≥ v

Si $\alpha < v < \beta$
→ -alpha-bêta (p-1, -β, -α) = v
→ alpha-bêta (p-1, -β, -α) = -v
→ min-max(p-1) = -v
→ -min-max(p-1) = v
→ min-max(p) = v
→ w = v

Si $v \geq \beta$
→ -alpha-bêta (p-1, -β, -α) ≥ v
→ alpha-bêta (p-1, -β, -α) ≥ -v
→ min-max(p-1) ≥ -v
→ -min-max(p-1) ≤ v
→ min-max(p) ≤ v
→ w ≤ v

On choisi de prendre -∞ et +∞ pour α et β respectivement.

Si alpha-bêta (p-1, -∞, +∞) = -∞
alors min-max(p) ≤ alpha-bêta (p-1, -∞, +∞) = -∞
donc min-max(p) = -∞ = alpha-bêta (p-1, -∞, +∞)

Si -∞ < alpha-bêta (p-1, -∞, +∞) < +∞
alors min-max(p) = alpha-bêta (p-1, -∞, +∞)

Si alpha-bêta (p-1, -∞, +∞) = +∞
alors min-max(p) ≥ alpha-bêta (p-1, -∞, +∞) = +∞
donc min-max(p) = +∞ = alpha-bêta (p-1, -∞, +∞)

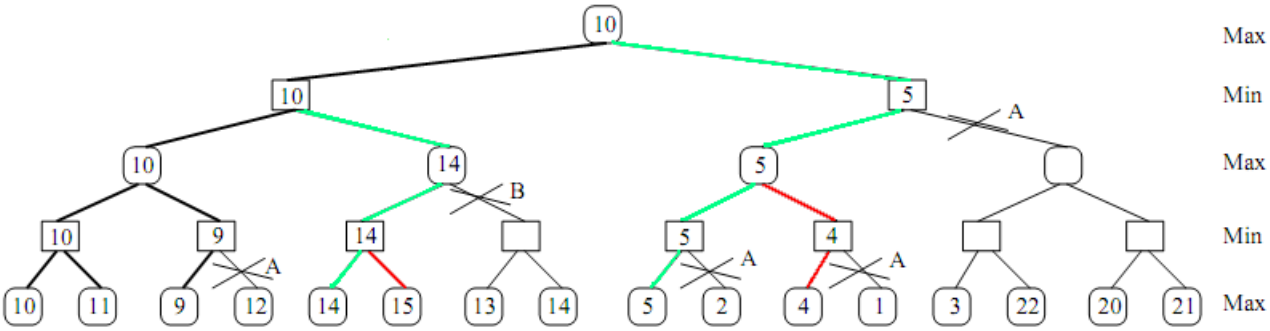
En conclusion, on prendra α = -∞ et β = +∞ pour que l'alpha-bêta ai le même résultat que le min-max tout en bénéficient de ça rapidité.

Initialisation et parcourt avec une fenêtre vide :

Souvent l'algorithme alpha-bêta doit effectuer un parcourt exhaustif d'une branche alors qu'il traite déjà des valeurs hors de la fenêtre alpha-bêta. L'élagage n'a pas lieu parce que les valeurs de alpha et de bêta sont transmises vers les feuilles car l'algorithme est en profondeur d'abord. Ainsi les nœuds parents doivent attendre que plusieurs branches soit développées pour que leurs paramètres soit actualisés et puissent réaliser une coupure. Dans tous les cas la première branche ne peut être élaguée c'est précisément pour cela qu'on sens sert pour l'initialisation.

La fenêtre vide ne fait que remonter la toute première feuille du développent et va provoquer une actualisation rapide d'alpha et de bêta avec cette feuille seulement. Il faut comprendre que le développement d'une fenêtre vide ne coute pratiquement rien. Si, par chance, la feuille d'une branche remontée par une fenêtre vide ne se trouve pas dans la fenêtre du nœud alors on ne la développera pas du tout. Si on n'a moins de chance, la feuille va tout de même resserre la fenêtre du nœud ce qui permettra de réaliser plus de coupure. Cette situation arrive ce produit avec une fréquence non négligeable.

Par exemple, cet arbre a subie plusieurs coupures. Si l'algorithme avait utilisé le principe des fenêtres vides (dont le parcourt est dessiné en vert), les trois branches rouge n'auraient pas étaient parcourues.



Entrées : profondeur (un entier), partie (une partie), alpha (un réel), bêta (un réel) et meilleur_coup (un coup)
Sorties : le meilleur_score que l'on puisse faire à une profondeur donné. Par ailleurs, le meilleur_coup sera actualiser.
Variables : meilleur_score (un réel), score (un réel).

```
début
{
   si le jeu est terminer OU que la profondeur est nulle alors
      retourner l'évaluation
   # On doit évaluer l'état de l'othellier

   si le joueur peut jouer alors
   {
      meilleur_coup ← premier_coup
      réaliser le premier coup
      changer de joueur
      meilleur_score ← -alpha-bêta (profondeur-1,-bêta,-alpha,NULL)
      changer de joueur
      annuler le premier coup
      # On n'initialise le meilleur_score (et si on est à la racine
      # le meilleur_coup aussi)

      si meilleur_score ≥ alpha alors
         alpha ← meilleur_score
      # On actualise la borne minimale alpha
```

```
            si meilleur_score < bêta alors
            # Si le meilleur_score dépasse la borne maximale bêta alors la
            # fenêtre alpha-bêta serait vide
            {
               pour la position légale suivante faire
               {
                  réaliser le coup un sur la position légale
                  changer de joueur
                  score ← -alpha-bêta (profondeur-1,-alpha - 1, -alpha, NULL)
                  # On parcourt déjà avec une fenêtre vide

                  si score est dans la fenêtre alpha-bêta alors
                     score ← -alpha-bêta (profondeur-1,-bêta,-alpha,NULL)
                  # On ne développe que si on n'est susceptible de trouver un
                  # score meilleur que meilleur_score

                  changer de joueur
                  annuler le coup joué

                  si score ≥ meilleur_score alors
                  {
                     meilleur_score ← score
                     meilleur_coup ← coup joué
                     # On actualise les meilleurs (attention pour la racine)

                     si meilleur_score > alpha alors
                     {
                        alpha ← score
                        # On doit resserrer la fenêtre

                        si meilleur_score ≥ bêta alors
                           retourner meilleur_score
                        # Si la fenêtre est vide on élague
                     }
                  }
               }
            }
         }
         sinon
         {
            changer de joueur
            meilleur_score = -alpha-bêta (profondeur-1,partie,-bêta,-alpha,NULL)
            changer de joueur
            # Le joueur ne peut pas jouer, on doit simplement retourner le score
            # de son adversaire comme s'il n'avait pas eu le trait !
         }

         retourner meilleur_score
      }
      fin
```

c. Tables de transpositions

Durant ce projet de nombreuses recherches ont été faites. Hélas, pour des raisons de temps certaines découvertes on était abandonnées à mon plus grand regret, notamment les tables de transpositions.

Les tables de transpositions sont la mémoire de l'algorithme alpha-bêta. En effet, lorsque l'on a choisi une des positions de jeu, il y a de grande chance qu'au tour suivant certaines ce ses positions soit encore jouable. Ainsi, si on a utilisé alpha-bêta, on a déjà réalisé un parcourt

pour ces positions. En réalité le problème et plus complexe et plus avantageux (déjà parce qu'il dépasse le simple stade d'une partie et pour d'autre raisons). Pour comprendre simplement disons que l'on va répertorier des suites de coups distincts et que pour chaque suite de coups on va mémoriser la profondeur de recherche, la borne alpha, la borne bêta et le meilleur coup à jouer.

Pour ce faire, on va utiliser une clé (un entier) qui sera représentatif de la suite de coups qui à été effectué sur l'othellier. Attention on utilise des suites de coups et non une configuration du damier : Deux configurations qui n'ont pas eu le même déroulement n'ont pas la même clé. Ainsi, dans notre alpha-bêta, avant de ce lancer dans une recherche on vérifie s'il on a des informations sur la clé de l'othellier et si ces informations sont assez précises. Cette notion de précision vient du fait que d'une profondeur à l'autre l'évaluation n'est pas la même ! Mais, si le cas se présente (et ça arrive souvent) à défaut de ne pas accepter le meilleur coup on pourra réduire la fenêtre alpha-bêta. Ainsi, au démarrage du programme on génère des nombres aléatoires pour toutes couleurs de toutes les cases de l'othellier plus une virtuel qui représentera le joueur ayant le trait. Remarque, on ne générera pas de nombre aléatoire pour la couleur vide, on lui affectera plutôt l'élément neutre du 'ou exclusif' : 0. Une clé sera le 'ou exclusif' de tous les nombres aléatoires de toutes les cases en fonction de la couleur qui l'occupe.

Cependant, un souci intervient rapidement : La mémoire ne pourra jamais conserver la totalité des enchainements d'une partie. En effet, l'intérêt qu'il y a générer autant de nombres aléatoires c'est de pouvoir construire une table de hachage bien répartie. On va donc créer une table de hachage d'une taille égal à une puissance de deux : 2^n. Ainsi, en ne considérant que les n derniers bits de la clé de l'othellier, on peut avoir accès au donné qui lui sont propre. Remarque, les autres bits serviront en cas de conflit.

IV. Fonction d'évaluation

On a vu que la fonction d'évaluation était un élément indispensable de l'algorithme de recherche. Pour évaluer efficacement le damier, il est nécessaire de choisir des critères judicieux tout en songeant à rapidité d'exécution de la fonction. Par ailleurs, chaque critères aura, à terme, un coefficient qui lui sera propre afin de refléter la valeur qu'il tient pour l'évaluation.

a. Les patterns

Les patterns sont des zones indépendantes de l'othellier (qui peuvent se recouper). L'idée fondamentale mis en œuvre pour utiliser des patterns est la suivante : On sauvegarde les issues des configurations observées d'un patterns durant un grand nombre de parties, puis, on cherche à reproduire les configurations gagnantes et on éviter les configurations perdantes. C'est de la bio-inspiration : On s'inspire de ce qui à marché pour marcher. Dans notre cas, on va s'inspirer d'une banque de sauvegarde de parties jouées entre de très bon joueurs d'Othello pour créer une mini base de donnés.

Les données de toutes les configurations qui seront sauvegardé dans un fichier par pattern sont :

- nombre de victoire
- nombre de nul
- nombre de partie
- points cumulés

A partir de ces quatre paramètres on définie la fonction de notations d'un pattern définie sur [-1 ; 1] par :

$$2.\frac{nombre\,de\,victoire + \frac{nombre\,de\,nul}{2} + \frac{1}{1+e^{-points\,cumulés}}}{nombre\,de\,partie+1} - 1$$

Remarque, cette fonction est bien le calcul qui permet de noter un pattern, mais comme tout critères d'évaluation, elle sera toujours associé à un coefficient propre au pattern : Un patterns un critère de positionnement.

Les patterns sont particulièrement intéressants car ils ne réalisent quasiment aucun calculs. Ils ne font que de rendre un nombre stocké en mémoire, ce qui est particulièrement rapide. Ainsi, après avoir réalisé l'apprentissage et sauvegardé les quatre paramètres, le programme pourra charger toutes ces données dans de grosses matrices à son démarrage. De ce fait, l'unique calcul qui devra être réalisé est celui de l'indice des matrices.

L'utilisation des patterns est une méthode élégante mais elle n'aura pas un grand intérêt si on ne peut associer une configuration à l'indice d'une matrice… On remarque que pour un patterns de taille n et du fait qu'une case n'a que 3 état possible, ce patterns aura en tout $3^n - 1$ configurations. Sachant que NOIR = 0, BLANC = 1 et VIDE = 2, on va donc pouvoir utiliser un système de numérisation en base 3.

Notre calcul ce présentera donc ainsi :

3^{n-1}*pattern[case 1] + … + 3^1*pattern[case n-1] + 3^0*pattern[case n]

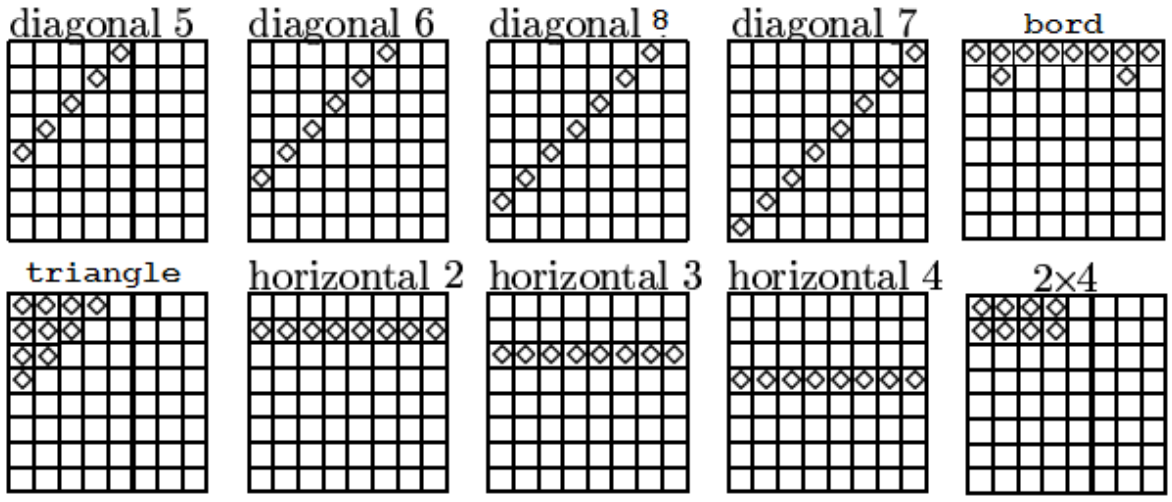
Soit après une factorisation par 3 :

3*(3*(3*(…(3*pattern[case 0]+pattern[case 1])…)))+pattern[case n]

Ce calcul, même s'il est unique devra être recalculer pour tout les patterns, mais on pourra parfois s'en économiser les frais. En effet, tous les patterns ne vont pas changer de configuration après à seul coup joué. C'est en fait que les patterns qui posséderont le pion posé et les pions retournés. C'est pourquoi, en associant chaque case de l'othellier aux patterns qui la contiennent on sélectionnera les indices qui devront effectivement être recalculer.

Les patterns correspondent donc à l'évaluation du positionnement des pions. Dans la fonction d'évaluation il seront tous sommé.

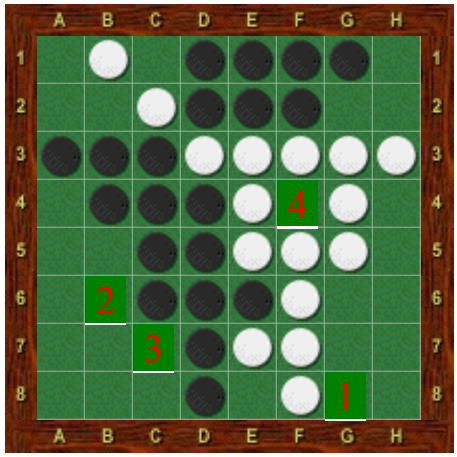
Voici les différents patterns utilisé par RODI (sans les symétries !) :



b. Mobilité potentielle

On a vu que la mobilité était un paramètre important dans l'élaboration d'une stratégie à Othello, il est donc naturel qu'il soit choisi comme critère d'évaluation.

Hélas, le calcul de la mobilité réelle et extrêmement couteux ! C'est pourquoi, RODI préfère approximer la mobilité globale par une somme de mobilité locale. Pour comprendre ce procédé rappelons comment est défini une position qui soit légale : Une position est légale s'il existe au moins une direction pour laquelle le pion posé encadre un ou plusieurs pions adverses avec un autre pion allié déjà placé sur le damier. De cette manière, une mobilité dite locale est une mobilité propre à deux directions opposées. Soit, la mobilité locale d'une case sera la somme de sa mobilité dans la diagonale descendante, dans la diagonale montante, dans la droite verticale et dans la droite horizontale.



Mobilité locale

c. Combinaison des critères

La fonction d'évaluation utilise les patterns pour calculer la 'note' d'une zone du damier. Ainsi, pour évaluer la totalité du damier on additionne toutes ces zones en leur attribuant leur coefficient. De cette façon, on ne considère que le critères de positionnement qui est nécessaire mais pas suffisant. On va donc ajouter la différence des pions des joueurs pour la maximisation, la différence des mobilités potentielles pour la mobilité et la différence du nombre de coin acquits (car de nombreux tests ont démontré son l'intérêt). Remarque, la parité n'a pas était retenu car elle est implicitement présente du fait du parcourt en profondeur réalisé.

Attention, une subtilité apparaît lorsque que l'on site la différence de critères entre les deux joueurs : Cette soustraction ne dépend en aucun cas du joueur qui à le trait ! En effet, les patterns on étaient calculés en fonction des parties gagné ou bien perdu pour un joueur fixé (pour RODI c'est le joueur NOIR). Il faut donc garder cette logique pour tout autre critère qu'il soit en établissant la règle suivante : On évalue toujours NOIR. Dans un second temps on se posera la question de qui à le trait et on modifiera en conséquence le signe de évaluation.

Remarquons que la mobilité potentiel, dépend de mobilité locale dont nous avons pas encore précisé quel était le calcul. En fait, les mobilités locales correspondant au mobilité de neuf patterns (et toutes leurs symétries) soit tous sauf le pattern 4x2 (et toutes ses symétries). Tout comme le positionnement, les mobilités locales sont pré-calculées. Au démarrage du programme, lors de remplir les matrices des patterns, le critère de mobilité associé à son coefficient et ajouté à la 'note' du patterns. Puisque tout les patterns seront ensuite sommé, la mobilité apparaitra avec la même valeur que si elle avait été additionné dans un second temps du fait de la linéarité des patterns. Ainsi, la fonction d'évaluation se contente de sommer les patterns (et implicitement la mobilité potentielle), la différence des matériels et la différence des coins acquits associé à leur coefficient. Remarque, la différence des matériels n'est pas associé à un coefficient. La linéarité de la fonction d'évaluation nous permet de fixer un unique coefficient (et les autres s'y ajusteront). Le choix du critère fixé et dû au fait que l'importance du critère de maximisation change au cours du temps d'une partie.

d. Découpage de l'évaluation

La fonction d'évaluation ne se comportera pas de la même manière en fonction de l'avancé dans la partie.

Si on se situe dans le début de la partie soit les 8 premier coups, la recherche sera à une profondeur un peu plus importante, mais surtout le coefficient fixé pour le critère maximisation sera négatif : Avoir beaucoup de pions en début de partie n'est pas avantageux.

Si on se situe dans le milieu de la partie soit pour un nombre de pions sur l'othellier entre 12 et 48. Dans ce cas de figure, le critère de maximisation ne sera plus un handicape. Il ne sera pas pour autant un bonus important.

Enfin, si on se situe dans les seize derniers coups, la profondeur explose : On va jusqu'à la fin de la partie et on n'évalue que la maximisation, soit le degré de victoire.

V. Apprentissage génétique

L'apprentissage génétique est encore une application de la bio-inspiration. Elle consiste à imiter la génétique pour optimiser une fonction. Dans le cadre du projet Othello, cet algorithme va nous permettre de retrouver les coefficients d'évaluation optimaux pour chaque critères.

a. Structures utilisée

Un individu :

On va obtenir un certain nombre de combinaison de coefficients que nous qualifierons d'ADN pour individu. Un individu sera donc représenté par son ADN mais aussi, ses phénotypes et son adaptation à l'environnement. Remarque, à l'initialisation les individus sont générés aléatoirement. L'adaptation d'un individu sera calculer en fonction des ses phénotypes selon la fonction de fitness suivante définie sur [0 ; 1] :

$$\frac{nombre\,de\,victoire + \frac{nombre\,de\,nul}{2} + \frac{1}{1+e^{-points\,cumulés}}}{nombre\,de\,partie + 1}$$

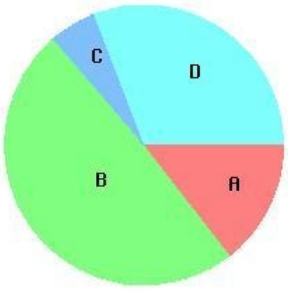
Une population :

Une population ce compose d'un échantillon mère, un échantillon fille et une génération. Un échantillon étant seulement un certain nombre d'individu. L'intérêt de cette structure réside dans le fait que lorsqu'une génération passe, la mère génère une fille. Ainsi, une fois l'échantillon fille calculé, il devient l'échantillon mère et la génération augmente. L'expérience sera renouvelé un très grand nombre de fois.

b. Algorithme utilisés

Pour générer l'échantillon fille à partir d'un échantillon mère, on va devoir effectuer trois opération : la sélection, le croisement et la mutation.

Sélection :



Selon la théorie de l'évolution de Darwin, un individu bien adapté à son environnement et amené se reproduire d'avantage qu'un individu mal adapté : C'est le fondement même de la sélection naturel. Sur le schéma B est l'individu le mieux adapté et C le moins bien.

Ainsi, la sélection des individus se fera selon une roue biaisé. Pour ce faire, on fixe une probabilité et un individu initial dans l'échantillon mère tout deux de manière aléatoire. On va ensuite tester un à un les individus et choisir le premier à avoir une adaptation supérieur à la probabilité fixé. Il se peut toutefois qu'il n'en existe pas ! Si le cas ce produit on prendra tout simplement l'individu initial.

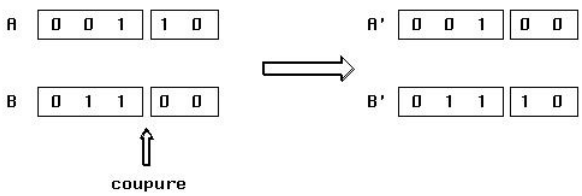
Entrée : population (une population)
Sortie : l'échantillon fille dispose des ADNS sélectionnés selon une roue biaisé. De plus tout leurs phénotypes son à zéro.
Variables : somme (un réel), probabilité (une probabilité) et premier (un individu)

```
début
{
   pour tout les individus de l'échantillon fille
   {
      probabilité ← un nombre aléatoire entre 0 et 1
      premier ← individu pris aléatoirement dans l'échantillon mère
      # on parcourt l'échantillon mère depuis un individu aléatoire
      compteur ← 0
      # on compte de nombre d'individu testé pour ne pas bouclé infiniment

      tant que adaptation de l'individu de la mère < probabilité
      ET compteur ≠ nombre d'individu
      {
         compteur ← compteur + 1
         individu suivant dans l'échantillon mère
      }

     recopier l'ADN de l'individu courant de l'échantillon mère dans celui
     de l'échantillon fille et mettre les phénotypes de ce derniers à zéro
   }
}
fin
```

Croisement :

Une fois la sélection faite, il est nécessaire de réaliser un croisement des ADNS. Le but n'est pas seulement de calquer jusqu'au bout la génétique, mais de couvrir d'avantage de configurations à partir de celles que l'on a généré aléatoirement.

Dans l'algorithme, on choisi pour un individu sur deux un coefficient aléatoirement sur son ADN. Puis on va le permuter avec celui coefficient de l'individu voisin et de même pour tous les autres coefficients jusqu'à la fin de l'ADN.



```
Entrée : fille (un échantillon)
Sortie : Les individus de l'échantillon fille sont croisés
Variables : coupure (un entier)

début
{
   coupure ← un nombre aléatoire entre 0 et le de nombre de coefficient

   pour tout individu de l'échantillon fille deux par deux a et b
   {
      pour le coefficient de la coupure jusqu'à la fin de l'ADN
         permuter le coefficient de l'individu a et celui de b
   }
}
fin
```

La mutation :

La mutation est un phénomène qui peut être particulièrement efficace pour notre optimisation. En effet, imaginons que la fonction d'évaluation présente plusieurs maximum locaux et admettons que nos échantillon converge vers l'un d'eux. Pour rompre la convergence et se donner une chance d'atteindre le maximum global, il est nécessaire de réaliser une mutation.

RODI fait muter chaque individu avec une probabilité 1/1000 (ce qui est faible). Remarque, si un individu mute, un seul de ses coefficient mute : Un ADN ne peut donc pas changer du tout au tout.

```
Entrée : fille (un échantillon)
Sortie : Les individu de l'échantillon fille sont mutés avec une
         probabilité de 1/1000
Variables : probabilité (une probabilité)

début
{
   pour tout individu de l'échantillon fille
   {
      probabilité ← un nombre aléatoire entre 0 et 1

      si probabilité ≤ 1/1000
         donner une valeur aléatoire à un coefficient aléatoire de l'ADN
   }
}
fin
```

c. Convergence des échantillons

L'apprentissage génétique de RODI calcul sans arrêt la descendances des échantillons. Mais du fait de la sélection, les échantillons vont converger vers une poignée d'ADN : Ce qui seront les mieux adaptés. Ces ADNS sont ceux qui sont susceptible de battre l'environnement. Si c'est le cas, on choisi le meilleur d'entre eux qui deviendra le nouvel environnement. Attention, un nouvel environnement nécessite de recalculer toutes les tables des patterns…

RODI tient compte d'un autre phénomène dû à la convergence des échantillons. Si les échantillons converge de génération en génération, alors à partir d'une génération suffisamment grande la moitié de l'échantillon est représentatif de ce dernier : On peut donc travail avec deux fois moins d'individus. A une génération encore plus grande, on pourra même ne considérer que le quart.

Cette réduction du nombres d'individu accélère considérablement l'apprentissage génétique ce qui permet à RODI de partir avec un grand nombre d'individu au départ : Un grand nombre d'individu généré aléatoirement multiplie les chances de trouver un bon ADN.

VI. Vue sur le programme

a. Aspect modulaire

Mon programme ce découpe en 10 parties :

→ Initialisations
→ Lectures/écritures
→ L'interpréteur
→ Règles du jeu
→ Recherche
→ Les patterns
→ Évaluation
→ Génétique
→ Constantes
→ Structures

b. Résultats observés

Suite à un certain nombre de contre temps, les résultats de RODI ne converges par vers la perfection… Les meilleurs démonstrations que RODI ai pu fournir est de battre AJAX en mode expert avec une probabilité de 0,125. Malheureusement le jeu de coefficients correspondant a été perdu.

Dés lors, la seule condition pour RODI de vaincre Ajax en mode expert et de retrouver des coefficients corrects. Deux ordinateurs on était réquisitionné dans mon entourage pour relancer l'apprentissage génétique de RODI : l'un en difficulté facile et l'autre en difficulté normale !

Malgré le doute qui règne, c'est avec zèle que défendrai RODI durant le tournoi.

c. Mode d'emploi

Voici les différentes commandes excitantes dans le programme et leur utilisation :

Commande	Argument	Action
new	Rien, 'easy', 'medium', 'hard'	lancer une nouvelle partie
game	position	jouer a la position donnée ou faire jouer l'IA
break		quitter la partie en cour
color		changer de couleur
undo		annuler un tour de jeu
redo		avancer un tour de jeu
load	fichier	charger une partie
save	fichier	sauvegarder une partie
help		afficher la liste des commandes
exit		quitter le programme

Observations personnelle

Lorsque le projet Othello m'a été remis, j'étais très déçu de devoir reprogrammer un jeu que j'avais déjà vu en première année. Après ces quelques moins de travaux, je reste déçu d'avoir eu ce projet en deuxième année de licence informatique car je ne pense pas avoir l'envergure d'un informaticien suffisamment expérimenté pour 'réellement' traiter les subtilités du jeu d'Othello qui ne sont pas mentionnées dans ce rapport. Je pense en fait que le premier rendu aurait du être notre point de départ. Néanmoins, ce projet m'aura permis de me lancer de multiple défies, de constater de multiples difficultés (notamment une panne d'ordinateur très mal venu et de toutes les pertes qu'elle puisse engendrer). Ce projet est donc pour moi une véritable expérience.