

Université de Bourgogne
1ère année de Master Informatique STIC
Parcours Informatique
Algorithme et complexité

JEU OTHELLO

Ghislain Loaec - Abdeldjalil Ramoul
2012-2013

Résumé

Ce rapport présente le travail effectué dans le cadre du projet d'algorithme et complexité

Table des matières

1	Introduction	2
2	Le jeu Othello	3
2.1	Principes du jeu	3
3	L'intelligence artificielle	4
3.1	Méthode aléatoire	4
3.2	Méthode du minimax	5
3.2.1	Algorithme du Minimax	7
3.2.2	Problème d'espace et de temps de recherche	9
3.2.3	Le Minimax à profondeur limitée	9
3.2.4	Le Minimax avec élagage <i>Alpha Beta</i>	11
4	Implémentation	14
4.1	Compilation	14
4.2	Utilisation	15
4.3	Algorithmes notables	16
4.3.1	Direction légale	16
4.3.2	Coup Légal	17
4.3.3	Simulation de jeu	17
4.3.4	Algorithme Minimax <i>Alpha Beta</i>	18
4.3.5	Algorithme : Tour de la machine de jouer	20
5	Module Othello : Documentation	21
6	Conclusion	24

Table des figures

1	Position de départ	3
2	Capture de jetons	4
3	Une partie de l'arbre de jeu de Othello	6
4	Exemple de déroulement de l'algorithme Minimax	9
5	Exemple de déroulement du Minimax à profondeur limitée avec d=1	10
6	Exemple de déroulement du Minimax à profondeur limitée avec d=2	11
7	Exemples de coupures alpha et beta	12

8	Exemple de déroulement de l'algorithme minimax alpha-beta avec $d = 2$	14
9	Utilisation du Makefile	15
10	Utilisation de l'exécutable	15
11	Configuration par défaut	16
12	Direction légale	16
13	Coup légal	17
14	Simuler un coup	18
15	Fonction récursive fold_until	19
16	Implémentation de l'algorithme minimax $\alpha\beta$	19
17	Méthode pour le tour de la machine	20

Liste des tableaux

1	Exemple de valeurs tactiques pour Othello	7
---	---	---

Liste des algorithmes

1	Algorithme Minimax	8
2	Algorithme Minimax à profondeur limitée	10
3	Algorithme Minimax avec élagage $\alpha\beta$	13

1 Introduction

Ce travail détaille le développement d'un simulateur du jeu Othello qui oppose un joueur à une intelligence artificielle. Il a été réalisé en se basant sur un algorithme de recherche du meilleur coup à jouer appelé Algorithme du minimax avec un élagage nommé élagage Alpha Beta. Nous allons présenter dans ce papier le jeu Othello et expliquer ces différentes règles. Dans la deuxième partie nous détaillerons la méthode du Minimax, ses limites et le principe de réduction de via un élagage alpha beta. Enfin nous présenterons le détail de l'application développée et l'implémentation des concepts vus dans la présentation de l'algorithme. Nous concluons sur les problèmes rencontrés dans l'avancement de ce projet et les suggestions apportées dans la vue d'une amélioration potentielle du logiciel.

2 Le jeu Othello

Othello est jeu de société à deux joueurs à information parfaite avec un nombre fini de stratégies. Dans cette catégorie de jeu, les joueurs jouent à tour de rôle et à chaque étape de la partie, le joueur dispose de toutes les informations sur l'état du jeu.

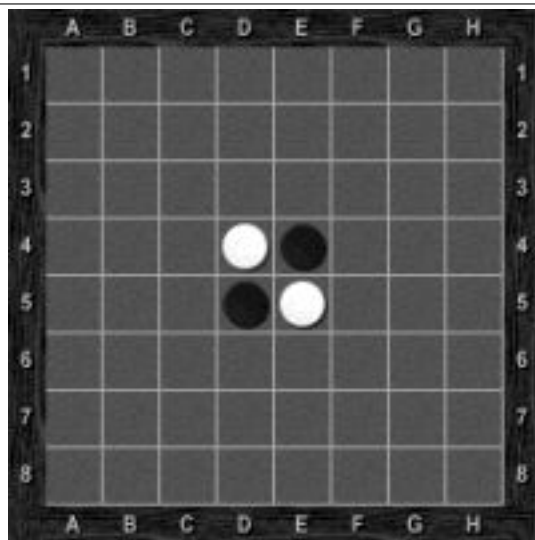
Une des caractéristiques qui rendent les jeux à information parfaite dignes d'étude, c'est que dans ce genre de jeu il existe toujours une solution optimale du fait de l'absence de l'incertitude à chaque mouvement. Néanmoins, ces solutions optimales sont souvent très coûteuses à calculer et peuvent être insoluble pour des jeux comme Othello. Donc l'utilisation de l'intelligence artificielle pour jouer à ce genre de jeu doit se reposer sur des heuristiques pour donner une approximation du jeu optimal.

2.1 Principes du jeu

Othello se joue sur un tablier unicolore de 64 cases (8 x 8) numérotées verticalement de 1 à 8 et Horizontalement de A à H. Les deux joueurs disposent de 64 jetons blancs d'un côté et noir de l'autre. Chaque couleur représente un des joueurs.

En début de partie deux jetons blancs sont posés sur les cases D4 et E5 et deux noirs sur les cases E4 et D5.

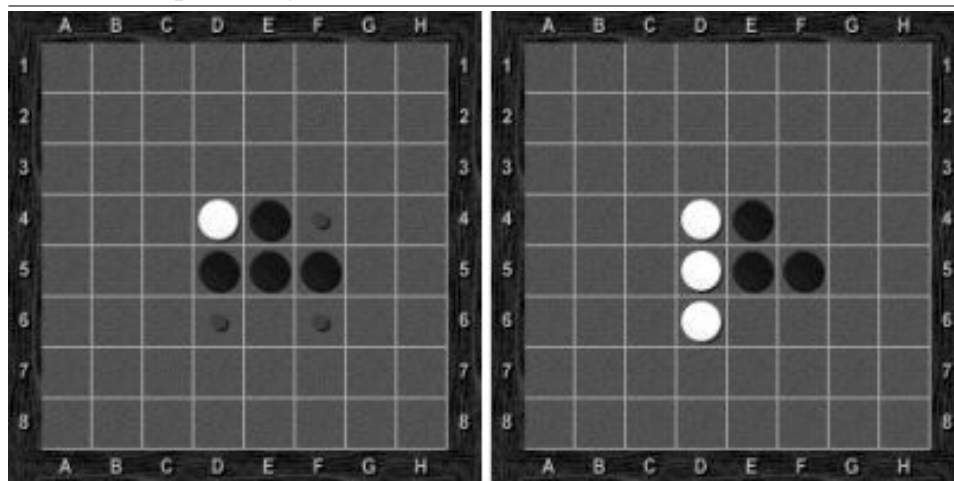
FIGURE 1 Position de départ



Les joueurs posent des jetons de leur couleur à tour de rôle selon des règles précises jusqu'à ce qu'aucun des deux ne puisse en mettre. A la fin on compte le nombre de jeton de chaque joueur, et c'est celui qui en a le plus qui gagne.

Le but du jeu est de capturer les jetons adverses afin de changer leur couleur. La capture de pions survient lorsqu'un joueur place un de ses jetons à l'extrémité d'un alignement de jetons adverses contigus et dont l'autre extrémité est déjà occupée par un de ses propres jetons. Le ou les pions embrassés par les deux jeton du joueur sont capturés.

FIGURE 2 Capture de jetons



Dans cet exemple on peut voir que le jour blanc peut mettre son jeton dans la case F4, D6 ou G6. En mettant son jeton dans la case D6 il capture le jeton D5 qui devient noir.

3 L'intelligence artificielle

Pour ce travail nous voulons opposer un joueur (humain) à une machine (intelligence artificielle). Pour cela, nous avons utilisé deux méthodes pour guider les choix de la machine :

3.1 Méthode aléatoire

Dans cette méthode, la machine répond aux actions du joueur par des choix de placements aléatoires parmi les positions possibles. Pour cela, elle

va choisir des cases du tablier de façon aléatoire et vérifier si elles sont possibles en se basant sur les règles du jeu. Cette méthode permet d'avoir l'illusion de jouer contre une machine, sauf que cette machine n'est guidée par rien pour prendre sa décision et ne peut être un adversaire efficace contre un joueur averti.

Pour rendre la machine plus intelligente il faut utiliser des méthodes qui évalueraient chaque coup et maximiseraient la probabilité de gagner. Une de ces méthodes consiste à utiliser l'algorithme minimax.

3.2 Méthode du minimax

La méthode du minimax est une technique de décision pour les jeux à deux joueurs basée sur le théorème du Minimax de Von Neumann dont l'objectif est de minimiser la perte maximale ou à l'inverse de maximiser le gain minimal pour un joueur.

En se basant sur le théorème fondamental de la théorie des jeux à deux joueurs, démontré en 1928 par John von Neumann. On sait que pour tout jeu à deux joueurs et à information parfaite avec un nombre fini de stratégies, il existe une évaluation V et une stratégie mixte (Choix aléatoire parmi une liste de possibilités) telle que :

- a) *Etant donné la stratégie du joueur A, le meilleur gain possible pour le joueur A est V , et*
- b) *Etant donné la stratégie du joueur B, le meilleur gain possible pour le joueur B est $-V$.*

On peut donc borner ses bénéfices et par la même occasion ceux de son adversaire. L'idée est de maximiser le gain minimum en minimisant le gain maximum de l'adversaire.

Avant d'aborder en détails l'algorithme du minimax, nous allons d'abord expliquer quelques notions qui nous aideront à mieux comprendre son fonctionnement.

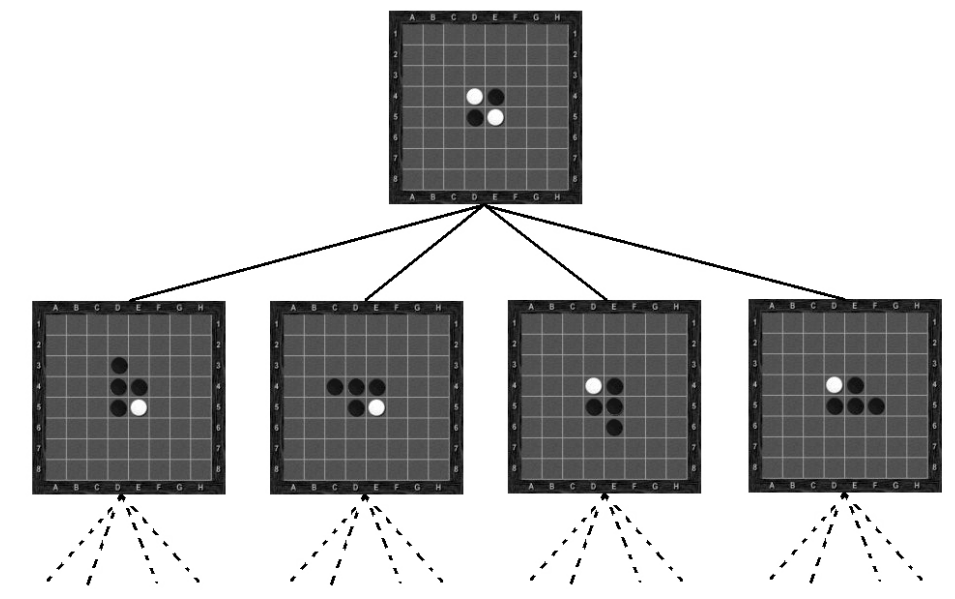
Les joueurs : Le joueur A est appelé Joueur Maximisant et le joueur B est appelé Joueur Minimisant. Dans ce travail le joueur maximisant est l'ordinateur, puisque c'est lui qui va exécuter l'algorithme et le joueur minimisant est son adversaire humain.

L'état du jeu : C'est une configuration possible du jeu. Par exemple dans Othello un état représente la disposition des jetons noirs et blancs dans l'Othellier. L'état initial représente l'état du jeu avant qu'il ait commencé. Et les états terminaux représentent les états qui mettent fin au jeu.

Le fils d'un état ei : noté $f(ei)$, il représente les états atteignables depuis ei .

L'arbre de jeu : C'est un arbre qui contient tous les états possibles du jeu. La racine représente l'état initial, et les feuilles représentent les états terminaux. Chaque noeud intermédiaire représente une position de jeu et chaque arc le reliant à son fils un coup possible permettant de passer à la position représentée par ce fils. La figure 3 représente une partie de l'arbre de jeu Othello engendré à partir de l'état initial.

FIGURE 3 Une partie de l'arbre de jeu de Othello



La génération de l'arbre de jeu est très couteuse parce que ce dernier est généralement gigantesque. Par exemple le nombre total d'états pour le jeu d'échecs est de 35^{100} . Donc on ne peut pas visiter tout l'arbre pour trouver les meilleurs coups à jouer.

- La fonction d'évaluation : Appelée aussi fonction heuristique, c'est une fonction qui associe à chaque noeud une valeur estimant les gains du coup pour le joueur maximisant. Par exemple, pour le jeu Othello plusieurs fonctions d'évaluation peuvent être choisies. On peut utiliser une matrice de valeurs tactiques associée aux cases pour avoir une fonction d'évaluation plus performante. Le tableau qui suit représente une matrice de valeurs tactiques en début de partie.

Dans les valeurs tactiques ci-dessus, nous constatons que les coins de

500	-150	30	10	10	30	-150	500
-150	-250	0	0	0	0	-250	-150
30	0	1	2	2	1	0	30
10	0	2	16	16	2	0	10
10	0	2	16	16	2	0	10
30	0	1	2	2	1	0	30
-150	-250	0	0	0	0	-250	-150
500	-150	30	10	10	30	-150	500

TABLE 1 – Exemple de valeurs tactiques pour Othello

l’othellier sont des positions très stratégiques puisque le gain estimé est de 500. Ceci s’explique par le fait qu’un jeton sur cette case est imprenable et offre la possibilité de capturer dans les mouvements suivant le maximum de jetons sur 3 axes différents. Il est également compréhensible que les cases qui entourent ces coins possèdent un gain négatif puis qu’il laisse une chance à l’adversaire de le capturer.

Dans ce travail nous avons choisi de ne pas considérer les valeurs tactiques cela nécessite une connaissance approfondie du jeu pour faire évoluer le tableau. Nous avons implémenté une fonction d’évaluation qui fait la différence entre le nombre de jetons du joueur et ceux de son adversaire. En plus d’être plus facile à programmer cette fonction d’évaluation laisse une chance de gagner aux joueurs non expérimentés comme nous.

3.2.1 Algorithme du Minimax

Le minimax est un algorithme pour la découverte de la solution optimale dans un jeu à deux joueurs à information parfaite. Il effectue une recherche en profondeur dans l’arbre de jeu pour décider du prochain coup à jouer. L’exploration des nœuds de cet arbre est limitée par un paramètre de profondeur. Pour tout cela il est nécessaire d’utiliser :

- Une fonction pour générer l’arbre de jeu, afin de déterminer les coups légaux à partir d’un état du jeu.
- Et une fonction heuristique pour évaluer un état de jeu.

à partir d’un état du jeu, l’algorithme visite l’arbre jusqu’à une profondeur préalablement définie. Il évalue ensuite les feuilles de l’arbre à l’aide de la fonction heuristique. Un score positif indique une bonne position pour le joueur A et un score négatif une mauvaise position, donc une bonne position pour le joueur B. Selon le joueur qui joue, le passage d’un niveau à

l'autre dans l'arbre est maximisant (pour le joueur A) ou minimisant (pour le joueur B). Chaque joueur essaie de maximiser son gain et de minimiser celui de son adversaire.

Le principe du minimax est de visiter l'arbre et de faire remonter la valeur minimax à la racine. La valeur minimax est calculée récursivement comme suit :

- $Minimax(e) = h(e)$, si e est une feuille de l'arbre (jeu terminé).
- $Minimax(e) = \max (Minimax(e1), \dots, Minimax(en))$, si e est un noeud Joueur (étape max)
- $Minimax(e) = \min (Minimax(e1), \dots, Minimax(en))$, si e est un noeud Adversaire (étape min)

Algorithme 1 Algorithme Minimax

Entrée: Noeud e

Sortie: Valeur Minimax du noeud e

Fonction MINIMAX(e)

Si final?(e) **Alors Retourner** $h(e)$

Sinon

Si joueur?(e) **Alors Retourner** $\max Minimax(ei) \mid ei \in f(e)$

Sinon Retourner $\min Minimax(ei) \mid ei \in f(e)$

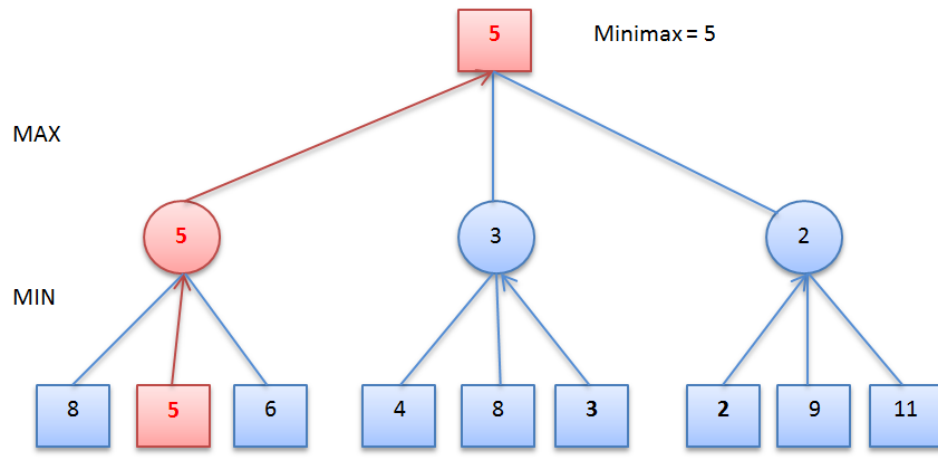
Fin Si

Fin Si

Fin Fonction

La figure 4 explique sur un exemple a deux niveaux le déroulement de l'algorithme du minimax. On peut voir que la racine ne contient pas le gain le plus élevé mais plutôt le maximum des pertes minimales. Donc l'algorithme ne cherche pas le score le plus élevé mais plutôt le gain le plus sûr.

FIGURE 4 Exemple de déroulement de l'algorithme Minimax



Propriétés de l'algorithme minimax

Pour l'algorithme minimax la complexité en temps est de $O(b^m)$ et la complexité en espace est $O(bm)$. Avec b = facteur de branchement et m = est la profondeur de l'arbre. Nous n'avons pas pu avoir des chiffres pour Othello mais par exemple pour les échecs $b = 35$ et $m = 100$.

3.2.2 Problème d'espace et de temps de recherche

Supposons qu'on veuille explorer tout l'arbre de jeu à partir de n'importe quelle position. Cela permettrait de déterminer s'il existe un chemin menant à la victoire, ou au moins au nul. Mais à cause du problème de l'explosion combinatoire, la génération d'un tel arbre est trop coûteuse pour être envisageable.

La première solution serait de limiter la profondeur d'exploration de l'arbre. Cela permettrait de réduire le nombre de coups et de réponses à évaluer. Dans ce cas on atteint rarement les feuilles sauf en fin de partie. Cette solution s'appelle le Minimax à profondeur limitée.

3.2.3 Le Minimax à profondeur limitée

C'est une variante de l'algorithme du minimax initial qui rajoute un indice de profondeur afin de limiter la taille de l'arbre à générer. Le principe de cette variante est de :

- Générer l'arbre de jeu jusqu'à une profondeur d à partir du noeud courant.

- Calculer la valeur de la fonction d'évaluation pour chaque noeud feuille, pas forcément terminal.
- Propager ces valeurs jusqu'aux noeuds non terminaux.

Algorithme 2 Algorithme Minimax à profondeur limitée

Entrée: Noeud e, Profondeur d

Sortie: Valeur Minimax du noeud e

Fonction MINIMAX(e, d)

Si final?(e) ou (d == 0) **Alors Retourner** h(e)

Sinon

Si joueur?(e) **Alors Retourner** $\max Minimax(e_i, d - 1) \mid e_i \in f(e)$

Sinon **Retourner** $\min Minimax(e_i, d - 1) \mid e_i \in f(e)$

Fin Si

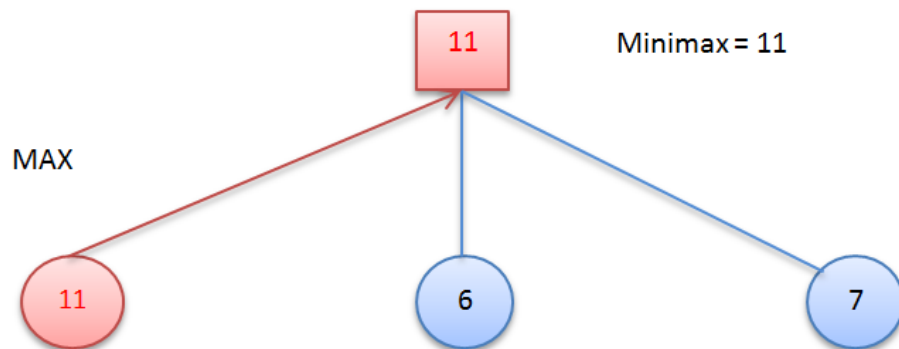
Fin Si

Fin Fonction

Limites de l'algorithme Minimax à profondeur limitée

Si la profondeur de recherche est trop petite. Cela veut dire que l'algorithme cherche à déterminer les coups immédiats qui maximisent le score du joueur A sans prévoir les coups de l'adversaire. La figure 5 montre le déroulement de l'algorithme Minimax à profondeur limitée avec d=1.

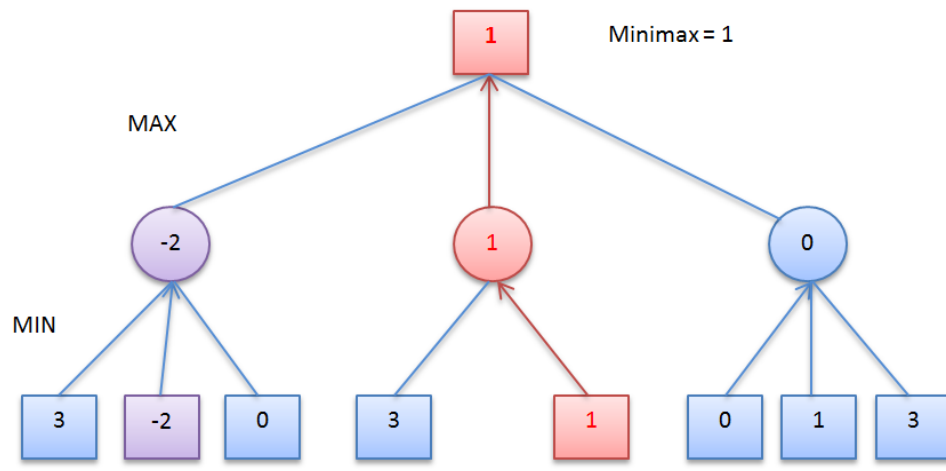
FIGURE 5 Exemple de déroulement du Minimax à profondeur limitée avec d=1



On peut voir que le programme détermine les coups légaux, simule le jeu et évalue chacun d'entre eux. Il décide ensuite que le coup avec l'évaluation 11 est le meilleur et le propage au sommet. Cette opération conduit

le programme à choisir le coup avec l'évaluation 11 ce qui peut s'avérer être une décision désastreuse puisque comme on peut le voir dans la figure 6, Le coup qui amenait à une position immédiate de score 11, va en fait amener la position du jeu à un score de -2. En effet si l'adversaire joue le coup qui mène vers l'évaluation -2, alors le score final serait -2. On s'aperçoit alors que le coup qui menait à l'évaluation 6 limite les dégâts avec un score de 1. Il sera donc préféré.

FIGURE 6 Exemple de déroulement du Minimax à profondeur limitée avec $d=2$



Le but est de maximiser la valeur de d pour prévoir le plus de coups possibles à l'avance. Mais sans pour autant être confronté au problème de l'explosion combinatoire. Pour cela on essaie d'élaguer l'arbre de recherche. On peut noter que dans la figure 6, il n'est pas nécessaire d'explorer toute la branche avec l'évaluation 0, puisque cette dernière est inférieure à 1 (on cherche le max). Le calcul des autres noeuds ne pourra pas améliorer la situation même si leurs scores sont meilleures que 0. La variante Alpha Beta du Minimax utilise cet élagage pour diminuer le nombre de branches à générer et permettre par la même occasion d'augmenter la valeur de d .

3.2.4 Le Minimax avec élagage *Alpha Beta*

L'élagage Alpha Beta est une méthode permettant de réduire la taille de l'arbre de jeu en élaguant les parties dont l'évaluation ne contribue pas à celle de la racine. Cette méthode augmente de manière significative les performances de l'algorithme Minimax sans affecter le résultat. Pour élaguer

certaines branches de l'arbre de jeu on définit deux bornes α et β tel que :

α représente de la borne inférieure de la valeur du noeud :

- $\alpha = h(e)$ sur les feuilles, et initialisée à $-\infty$ ailleurs.
- Dans les noeuds joueurs, $\alpha = \text{MAX}$ des valeurs obtenues sur les fils visités jusque-là.
- Dans les noeuds adversaires, elle est égale à la valeur α de son prédécesseur.

β représente de la borne supérieure de la valeur du noeud :

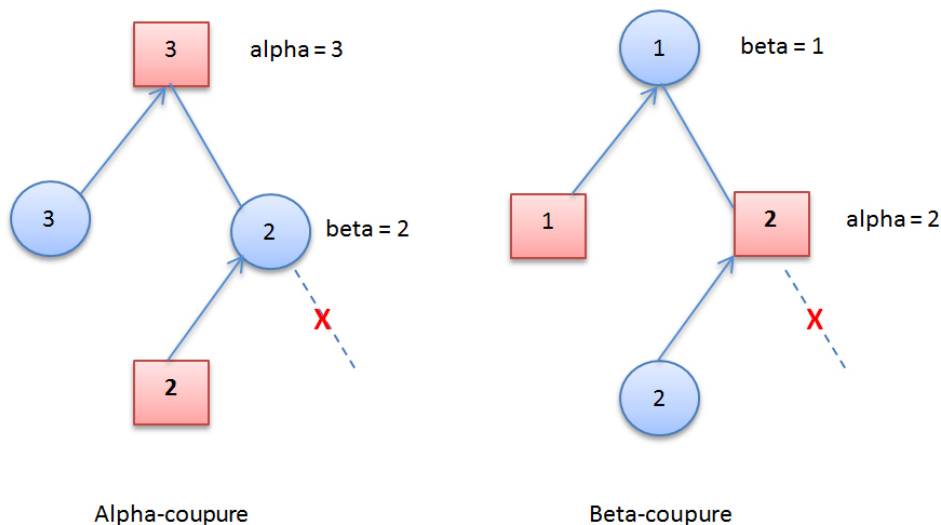
- $\beta = h(e)$ sur les feuilles, et initialisée à $+\infty$ ailleurs.
- Dans les noeuds adversaires, $\beta = \text{MIN}$ des valeurs obtenues sur les fils visités jusque-là.
- Dans les noeuds joueurs elle est égale à la valeur β de son prédécesseur.

Pour Othello, nous avons remplacé les valeurs $-\infty$ et $+\infty$ respectivement par les valeurs -64 et +64, puisque dans Othello la différence max entre les jetons des joueurs (fonction heuristique) ne peut dépasser 64. Les noeuds élagués sont ceux tel que

$$H(e) \in [\beta, \alpha] \text{ et } \alpha \geq \beta$$

Donc l'alpha-coupure intervient dans un noeud adversaire (MIN) lorsque sa beta-valeur est inférieure à l'alpha-valeur de son parent. Et la beta-coupure intervient dans un noeud joueur (MAX) lorsque sa alpha-valeur est supérieure à la beta-valeur de son parent.

FIGURE 7 Exemples de coupures alpha et beta



Algorithme 3 Algorithme Minimax avec élagage $\alpha\beta$

Entrée: Noeud e , Profondeur d , Alpha α , Beta β

Sortie: Valeur Minimax du noeud e

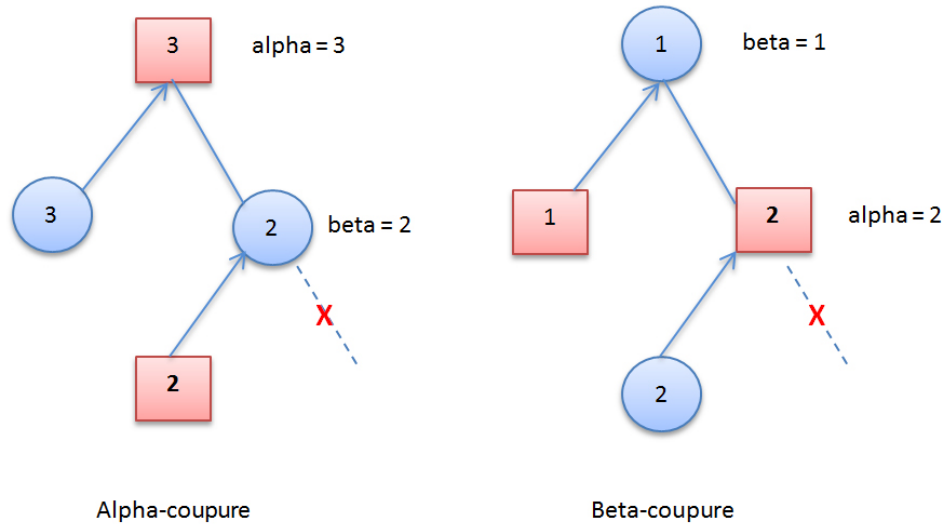
```
Fonction MINIMAX( $e, d, \alpha, \beta$ )  
  Si final?( $e$ ) ou ( $d == 0$ ) Alors Retourner  $h(e)$   
  Sinon  
    Si joueur?( $e$ ) Alors  
       $v = -\infty$   
      Pour  $fi \in f(e)$  Faire  
         $v = \max(v, \text{Minimax}(fi, d - 1, \alpha, \beta))$   
        Si  $v > \beta$  Alors Retourner  $v$   
      Fin Si  
       $\alpha = \max(\alpha, v)$   
    Fin Pour  
  Sinon  
     $v = +\infty$   
    Pour tout les  $fi \in f(e)$  Faire  
       $v = \min(v, \text{Minimax}(fi, d - 1, \alpha, \beta))$   
      Si  $v < \alpha$  Alors Retourner  $v$   
    Fin Si  
     $\beta = \min(\beta, v)$   
  Fin Pour  
  Fin Si  
Fin Fonction
```

Dans la figure suivante nous observons l'exécution de l'algorithme du minimax avec élagage alpha beta sur l'arbre de jeu vu dans la figure 6.

On peut observer que le résultat reste le même que celui obtenu en appliquant l'algorithme du minimax classique. Mais l'élagage alpha beta a permis de ne pas visiter les dernières branches, ce qui augmente les performances de l'algorithme et nous permet par la même occasion d'augmenter la profondeur d'exploration du graphe. L'ordre d'apparition des noeuds influence beaucoup le rendement de l'élagage. Examiner d'abord les noeuds à forte valeur d'évaluation maximise le rendement de l'algorithme.

La complexité en temps de l'algorithme minimax avec élagage alpha beta revient dans le meilleur des cas à $O(b^{(m/2)})$. Cela permet d'augmenter la profondeur de recherche pour améliorer les résultats sans trop affecter les performances. Et dans le pire des cas elle est identique à celle du

FIGURE 8 Exemple de déroulement de l'algorithme minimax alpha-beta avec $d = 2$



minimax.

4 Implémentation

Dans cette partie nous allons détailler les étapes d'implémentation de l'application.

4.1 Compilation

Afin de générer les fichiers binaires, nous avons créé un Makefile très générique paramétrable à souhait. Il semblerait que les performances d'exécution soit optimisées en utilisant le compilateur `ocamlc`. Ce qui rend le choix intéressant dans un cas d'utilisation comme le nôtre ou le programme est sujet à des explosions combinatoires.

Pour une compilation standard avec `ocamlc`, utiliser : `make`

FIGURE 9 Utilisation du Makefile

```
# Pour recompiler le système progressivement :
#     make
# Pour recalculer les dépendances entre les modules :
#     make depend
# Pour supprimer l'exécutable et les fichiers compilés :
#     make clean
# Pour compiler avec le compilateur de code natif
#     make opt
```

4.2 Utilisation

Afin de lancer l'application, tapez : `./othello`

Il est possible de spécifier des arguments à l'exécutable qui vont influencer le déroulement du jeu. Vous pouvez obtenir la liste ces options avec :

FIGURE 10 Utilisation de l'exécutable

```
$ ./othello --help
othello
  -size <int> : Taille d'une case en pixels
  -ia <bool>  : Intelligence artificielle on/off
                | true  -> Minimax ab
                | false -> Case aléatoire
  -depth <int> : Profondeur maximale d'exploration
                de l'arbre des possibilités Minimax
  -background <int> <int> <int> : Couleur du fond (RGB)
  -help : Afficher cette liste d'options
  --help : Afficher cette liste d'options
```

Les options par défaut de l'application sont :

FIGURE 11 Configuration par défaut

```
1 | val cell_size : int ref = {contents = 50}
2 | val bg_r : int ref = {contents = 50}
3 | val bg_g : int ref = {contents = 150}
4 | val bg_b : int ref = {contents = 50}
5 | val size : int ref = {contents = 8}
6 | val ia : bool ref = {contents = true}
7 | val depth : int ref = {contents = 4}
```

4.3 Algorithmes notables

4.3.1 Direction légale

Cette méthode permet de déterminer si une direction de jeu est légale. À partir d'une case donnée, l'algorithme récursif initie la direction comme illégale. En regardant le premier voisin dans la direction donnée, la case doit contenir impérativement un jeton du joueur opposé. L'itération récursive continue alors en indiquant grâce à un booléen qu'au moins un jeton de l'adversaire est dans l'alignement. Lorsqu'un jeton de couleur de joueur est identifié, l'algorithme s'arrête et la direction est jouable. Si une case vide est rencontrée, la direction n'est pas légale.

FIGURE 12 Direction légale

```
1 | (* Methode de test de direction légale *)
2 | let playable_dir board c (x, y) (dx, dy) =
3 |   let rec playable_dir_rec (x, y) valid =
4 |     if not (check_pos board x y) then
5 |       false
6 |     else (
7 |       match board.(x).(y) with
8 |       | Empty -> false
9 |       | cell ->
10 |         if cell = (get_opponent c) then
11 |           playable_dir_rec (x + dx, y + dy) true
12 |         else
13 |           valid
14 |     )
15 |   in playable_dir_rec (x + dx, y + dy) false
16 | ;;
```

4.3.2 Coup Légal

Afin de déterminer si un coup est légal, cet algorithme va tester si parmi les 8 directions possibles de capture, au moins l'une d'être est jouable. Pour ce faire et à partir de la liste des directions donnée, un fold-left testera chacune d'entre elles et comparera leur légalité grâce à un simple prédicat. Nous aurions pu optimiser légèrement cette fonction en utilisant notre fonction 'fold-until' décrit dans la section 'Implémentation Minimax Alpha Beta' et stopper le test des directions à partir du moment où l'une d'entre elle est légale.

FIGURE 13 Coup légal

```
1 (* Methode de test de coup légal => Vrai si le coup est légal *)
2 let playable_cell board c x y =
3   if not (check_pos board x y) then
4     false
5   else (
6     let directions = [
7       (-1, -1); (-1, 0); (-1, 1);
8       (0, -1); (* X *) (0, 1);
9       (1, -1); (1, 0); (1, 1)
10    ]
11    in match board.(x).(y) with
12    | Empty -> ( true && (
13      List.fold_left
14        (fun a b -> a || b)
15        false
16          (List.map
17            (fun d -> playable_dir board c (x, y) d)
18            directions
19          )
20        )
21    | _ -> false
22  )
23 ;;
```

4.3.3 Simulation de jeu

Cette méthode permet la simulation d'un coup joué. Tout premièrement, l'algorithme va copier l'état du tablier et la position des jeton. Le tableau contenant uniquement des références sur jetons (type cell), il nous est impossible de copier le tableau via un Array.copy par exemple, puisque les références copiées pointeront toujours sur les mêmes objets de case. Nous utiliserons donc une fonction customisée de copy de tableau. L'algorithme

récuratif de capture des jetons est très similaire à celle vu précédemment. La fonction retourne le plateau de jeu à l'état suivant le coup simulé.

FIGURE 14 Simuler un coup

```
1  (* Méthode pour simuler le jeu sur une case *)
2  let sim_play_cell board c x y =
3    let sim_board = (copy_board board) in
4    let directions = [
5      (-1, -1); (-1, 0); (-1, 1);
6      (0, -1); (* X *) (0, 1);
7      (1, -1); (1, 0); (1, 1)
8    ]
9    and opponent = (get_opponent c)
10   in (
11     List.iter
12       (fun (dx, dy) ->
13         if (playable_dir sim_board c (x, y) (dx, dy)) then
14           let rec take (x, y) =
15             if (check_pos sim_board x y) then
16               if (sim_board.(x).(y) = opponent) then (
17                 sim_board.(x).(y) <- c;
18                 take (x + dx, y + dy)
19               )
20             in take (x + dx, y + dy)
21           )
22         directions
23       );
24     sim_board.(x).(y) <- c;
25     sim_board
26   );;
```

4.3.4 Algorithme Minimax *Alpha Beta*

Dans un premier temps, le langage OCAML ne permettant pas d'interrompre l'exécution d'une boucle directement via des instructions telles que 'break', 'return', nous avons dû créer une fonction récursive (figure 15) permettant d'itérer à travers une liste d'éléments en intégrant un accumulateur et un callback. L'itération s'arrêtera lorsqu'un prédicat [p] sur l'accumulateur sera validé.

Nous pourrions alors facilement implémenter l'algorithme Minimax $\alpha\beta$ détaillé précédemment.

FIGURE 15 Fonction récursive `fold_until`

```
1  (** Méthode récursive de Fold left sur une liste avec la fonction [f]
2     jusqu'à que ce que le prédicat [p] soit satisfait *)
3  let rec fold_until f p acc l =
4      match l with
5      | t :: q when p acc -> acc
6      | t :: q -> fold_until f p (f acc t) q
7      | [] -> acc
8  ;;
9  (** Retourne l'accumulateur *)
```

FIGURE 16 Implémentation de l'algorithme minimax $\alpha\beta$

```
1  (** Méthode récursive de calcul alphabeta des noeuds de l'arbre *)
2  let rec alpha_beta board c d a b =
3      if (is_finished board or d = 0) then
4          score board White
5      else let playable_cells = playable_cells board c in
6          match c with
7          | White -> let a2 = ref a in
8              fold_until
9              (
10                 fun v (x, y) ->
11                     let ab = alpha_beta
12                         (sim_play_cell board c x y)
13                         Black
14                         (d - 1)
15                         !a2 b in
16                     let v2 = max v ab in
17                     a2 := max v2 !a2;
18                     v2
19             )
20             (fun v -> v > b)
21             (-(Array.length board) * (Array.length board.(0)))
22             playable_cells
23          | _ -> let b2 = ref b in
24              fold_until
25              (
26                 fun v (x, y) ->
27                     let ab = alpha_beta
28                         (sim_play_cell board c x y)
29                         White
30                         (d - 1)
31                         a !b2 in
32                     let v2 = min v ab in
33                     b2 := min v2 !b2;
34                     v2
35             )
36             (fun v -> v < a)
37             ((Array.length board) * (Array.length board.(0)))
38             playable_cells
39  ;;
```

4.3.5 Algorithme : Tour de la machine de jouer

Cette méthode décrit le tour de la machine. À partir de la liste des coûts immédiats, l'algorithme va évaluer chacun d'entre eux grâce à l'algorithme Minimax $\alpha\beta$. Les coup sont alors pondérés et la fonction récursive va retourner le meilleur mouvement à faire. Le programme va alors jouer la case grâce au coordonnées de la position de cette dernière.

FIGURE 17 Méthode pour le tour de la machine

```
1  (** Methode: Tour de la machine intelligente *)
2  let ia_turn board =
3  let playable_cells = playable_cells board White in
4  match (List.length playable_cells) with
5  | 0 -> ()
6  | _ ->
7      (let x, y =
8         let rec get_best_move ab cell playable_cells =
9             match playable_cells with
10             | (x, y) :: q ->
11                 let a = -((Array.length board) * (Array.length board.(0)))
12                 and b = ((Array.length board) * (Array.length board.(0)))
13                 and old_ab = ab in
14                 let ab = (
15                     alpha_beta
16                     (sim_play_cell board White x y)
17                     Black
18                     (!depth - 1)
19                     a b
20                 ) in
21                 if ab > old_ab then get_best_move ab (x, y) q
22                 else get_best_move old_ab cell q
23             | [] -> cell
24         in get_best_move
25             (-((Array.length board) * (Array.length board.(0))))
26             (List.hd playable_cells)
27             (List.tl playable_cells)
28         in play_cell board White x y)
29  ;;
```

5 Module Othello : Documentation

```
val cell_size : int Pervasives.ref
    Définition de la largeur des cellules en pixels

    Défaut : 75

val bg_r : int Pervasives.ref
    Niveau de rouge de couleur du fond : 0:255

    Défaut : 50

val bg_g : int Pervasives.ref
    Niveau de vert de couleur du fond : 0:255

    Défaut : 150

val bg_b : int Pervasives.ref
    Niveau de bleu de couleur du fond : 0:255

    Défaut : 50

val size : int Pervasives.ref
    Nombre de cases qui constitue l'arrête du plateau de jeu

    Défaut : 8

val ia : bool Pervasives.ref
    Utilisation ou non de l'intelligence artificielle

    Défaut : true (Activée)

val depth : int Pervasives.ref
    Profondeur d'exploration de l'arbre de coups légaux

    Défaut : 4

type cell =
  | White
      Jeton blanc
  | Black
      Jeton noir
  | Empty
      Case vide
    Valeurs possibles d'une case
```

```

type board = cell array array
    Matrice représentative du tablier : tableau de cases à 2 dimensions

type coord = int * int
    Représentation d'un position par ses coordonnées

type coord_list = coord list
    Liste représentative de positions

val make_board : cell array array
    Méthode de construction du plateau

    Retourne un tablier de case vides

val copy_board : 'a array array -> 'a array array
    Méthode de copie d'un état du tablier

    Retourne une matrice avec les références des nouvelles case

val init_board : cell array array
    Méthode d'initialisation des positions des jetons lors d'une nouvelle
    partie

    Retourne un tablier avec 4 jetons positionés sur les cases D4 E4 D5 E5

val display_cell : cell array array -> int -> int -> unit
    Méthode d'affichage d'une case

val display_board : cell array array -> unit
    Méthode d'affichage du plateau de jeu

val display_message : string -> unit
    Méthode d'affichage des messages

val count : 'a array array -> 'a -> int
    Méthode pour compter le nombre de jeton d'une couleur donnée sur
    un état donné

val is_finished : cell array array -> bool
    Méthode de test de fin de partie

val check_pos : 'a array array -> int -> int -> bool
    Méthode de test de position => Vrai si sur le plateau

```

```

val get_opponent : cell -> cell
    Methode de recuperation de la couleur de l'adversaire

val playable_dir : cell array array -> cell -> int * int -> int * int -> bool
    Methode de test de direction legale => Vrai si la direction est legale

val playable_cell : cell array array -> cell -> int -> int -> bool
    Methode de test de coup legal => Vrai si le coup est legal

val play_cell : cell array array -> cell -> int -> int -> unit
    Methode pour jouer une case

val sim_play_cell :
    cell array array ->
    cell -> int -> int -> cell array array
    Methode pour simuler le jeu sur une case

    Retourne la matrice de references sur les nouvelles positions de jetons

val playable_cells : cell array array -> cell -> (int * int) list
    Methode de recuperation de la liste des coups jouables

    Retourne une liste des coordonnees des coup legaux

val score : cell array array -> cell -> int
    Methode qui retourne le score pour un etat de jeu et un joueur

val display_scores : cell array array -> unit
    Methode pour afficher les scores

val fold_until : ('a -> 'b -> 'a) -> ('a -> bool) -> 'a -> 'b list -> 'a
    Methode recursive de Fold left sur une liste avec la fonction f
    jusqu'à que ce que le predicat p soit satisfait

    Retourne l'accumulateur

val alpha_beta : cell array array -> cell -> int -> int -> int -> int
    Methode recursive de calcul alphabeta des noeuds de l'arbre

val ia_turn : cell array array -> unit
    Methode : Tour de la machine intelligente

val rdm_turn : cell array array -> unit

```


Methode : Tour de la machine aléatoire

```
val player_turn : cell array array -> unit  
Methode : Tour du joueur
```

```
val end_message : cell array array -> string  
Méthode d'affichage du message en fin de partie
```

```
val continue : unit -> bool  
Méthode d'attente d'un événement click de souris
```

```
val game : unit -> unit -> unit  
Methode de définition d'une partie
```

```
val speclist : (string * Arg.spec * string) list  
Définition des spécifications et des arguments possibles
```

```
val main : unit -> unit -> unit  
Définition de la fonction principale
```

6 Conclusion

Ce projet nous a permis de mettre en application les enseignements d'Algorithmes et complexité de ce semestre. L'approche du problème avec le langage fonctionnel OCAML a introduit les notions de programmation impérative (même si pas le cas échéant) et de filtrage par motif qui rend facile la manipulation de types algébriques. Ce projet a dissipé toute appréhension à travailler avec ce langage multi-paradigme. Ceci nous a également permis de nous documenter sans limites concernant les intelligences artificielles et assimiler leurs différents concepts.