Institut Galilée

INFO 2 :
Stéphane FOURNIER
Yohan ROUSSET

Programmation Logique

Othello

Année 2009-2010

Enseignante : C.Rouveirol

Sommaire

I. Introduction

Dans le cadre de notre formation d'ingénieur, nous sommes amenés à suivre un cours de Programmation Logique. A travers ce cours, nous abordons l'intelligence artificielle avec toutes les notions inhérentes à ce sujet. Nous avons alors été amenés à implémenter le jeu Othello afin de manipuler ces notions. C'est alors que nous utiliserons un langage adapté à la programmation logique : Prolog.

Nous verrons donc dans un premier temps les principales caractéristiques de notre implémentation en analysant les structures de données choisies et les différents algorithmes que nous avons utilisé. Puis, nous décrirons les différents joueurs que nous aurons implémentés en expliquant les caractéristiques de chacun. Enfin, nous terminerons ce document par une courte liste d'améliorations qu'il est possible d'effectuer afin de rendre notre joueur heuristique plus fort que ce qu'il est aujourd'hui.

II. L'implémentation

Dans cette partie nous nous intéresserons au codage du joueur aléatoire et du joueur heuristique, en évoquant les choix des structures de données, des algorithmes ou encore des différentes heuristiques mises en place. Une fois cela terminé, nous pouvons alors explorer l'arbre de jeux et prévoir les meilleurs coups à l'aide de l'algorithme alpha-beta.

A. Structures de données

1. Définition de la structure du plateau

Pour modéliser notre plateau de jeu Othello, nous avons choisi d'utiliser une liste de six listes, la première liste représente le plateau de jeux alors que les six autres listes représentent les lignes du plateau. Il nous a paru plus facile pour la suite du programme de choisir cette structure car elle présente l'avantage d'être plus lisible qu'une simple liste de 36 éléments. En outre, d'après nos tests, l'implémentation de nos fonctions d'accès et de placement de pion dans les cases du plateau se fait très rapidement. Ainsi, l'accès se fait en seulement 2 inférences, ce qui est selon nous très performant.

Pour modéliser les pions nous avons choisi d'utiliser les types :

- « blanc » et « noir » pour les pions présent sur le plateau ;
- « _ » (variable) pour les cases vides du plateau.

Le problème amené par cette structure survient lorsque nous devons « flipper » des pions. En effet, il faut pour cela recréer une grille de jeu vide (via le prédicat *preparePlateau*/1). Puis, recopier dans cette grille les modifications apportées par le prédicat *flipper*, avant de recopier les précédentes valeurs non modifiées à partir du plateau source. Mais nous avons constaté qu'en temps CPU, cela n'était pas long.

2. Amélioration possible du programme

Une autre solution aurait consisté à créer deux prédicats : b(X), et n(X), représentant respectivement le blanc et le noir, si X est une variable libre. Ainsi, pour flipper un pion, il suffit d'instancier la variable X à la valeur de son choix.

Seulement nous avons pensé que l'implémentation de cette structure aurait pu être rapidement lourde si un pion est de nombreuse fois flippé durant une partie (c'est le cas pour les pions situés au centre du plateau de jeu). En effet, pour trouver la couleur d'une case, il vaut aller

observer le prédicat le plus intérieur. Or un pion qui a changé de nombreuse fois de couleur (*b(n(b(n(b(n(b(X)))))))* représente un pion blanc) peut pousser le programme à prendre beaucoup de temps CPU pour déterminer sa couleur.

C'est pourquoi nous avons décidé de choisir la structure définie précédemment, puisqu'elle propose, selon nous, une meilleure utilisation du temps CPU.

3. Accès et modification de la valeur d'une case.

Afin de facilité l'accès au plateau de jeux, nous avons défini des prédicats *acces* et *accesLigne*. Le premier prédicat permet de sélectionner une ligne du plateau donnée en paramètre. Ce prédicat appellera alors *accesLigne* avec la colonne spécifié en argument de *acces* et unifiera le dernier paramètre de *acces* avec le type du pion posé sur le plateau (blanc, noir ou libre).

De façon analogue, deux prédicats *place* et *placeLigne* ont été définis. Ils permettent d'unifier une variable du plateau de jeux, à une ligne et une colonne données, à une valeur donnée en paramètre (typiquement *blanc* ou *noir*). Le prédicat échoue si la case du plateau de jeu n'est pas une variable libre. Il faut donc au préalable vérifier avec *acces* que la case contient bien une variable.

B. Algorithme

Dans un premier temps, nous avons mis en place le plateau de jeu avec un joueur aléatoire. Pour cela, il nous a fallu trouver un premier algorithme qui permet de trouver tous les coups jouables sur un plateau donné puis un autre pour poser un pion. Afin d'améliorer l'efficacité du programme, nous avons choisi de développer une heuristique de jeu. Un autre algorithme intervient dans l'amélioration du programme, il s'agit de l'élagage alpha beta.

1. Tous les coups jouables

Afin de déterminer, pour un plateau de jeu donné et une couleur de pion (un joueur en fait) donnée, les coups possibles, nous avons conçu un prédicat *tous_coups_legaux/3* qui prend trois paramètres. Ainsi, pour une couleur (*noir* ou *blanc*), et un plateau de jeu, il renvoie une liste de coups légaux. Il est à savoir qu'un coup est formaté de la forme *coup(NuméroLigne,NuméroColonne).*

En fait, ce prédicat se sert d'un prédicat auxiliaire qui renvoie un coup légal possible : *position_est_jouable/4.* Ce prédicat à 4 arguments est vrai, pour un plateau donné, une couleur en particulier, et un numéro de ligne et un numéro de colonne, si la case référencée par la ligne et la colonne choisie est jouable. Pour déterminer si une case peut recevoir un pion, elle doit etre libre (ce

doit etre une variable donc), puis le prédicat lance des vérifications dans toutes les directions possibles (haut, bas, gauche, droite et les 4 diagonales). Les 8 prédicats de vérifications vérifient que la première case située dans la direction à vérifier est bien de la couleur opposée au joueur demandant ses coups légaux. Puis elle vérifie qu'il existe bien une série de pions continue (sans case vide), jusqu'à un pion de sa propre couleur. On vérifie en fait qu'il existe bien un « sandwich ». Pour plus de détail, il est recommandé de consulter le code d'un des joueurs implémentés.

Pour avoir tous les coups possibles, il suffit de se servir du mécanisme de backtrack offert par Prolog. Ainsi, *tous_coups_legaux/3* utilise en fait un *findall* afin de récupérer tous les indices de lignes et de colonnes admettant un coup jouable. Le prédicat est alors utilisé ainsi : *position_est_jouable(+Position,+Couleur,-IndiceLigne,-IndiceColonne)*.

Il a été créé un prédicat *move/3* qui fonctionne exactement de la même manière que tous_coup_legaux. C'est ce prédicat qui est utilisé dans les algorithmes de recherches que nous verrons plus tard.

2. Jouer un coup

Le but du jeu d'Othello est d'avoir, à la fin de la partie, le plus de pion de sa couleur possible (ou en tout cas, pour gagner, il en faut plus que son adversaire). Or, pour atteindre ce but, il faut avant toute autre chose savoir poser un pion.

En effet, on peut se poser la question suivante : comment jouer un coup ?

Il existe alors deux possibilités. Soit le joueur courant est un joueur de type aléatoire, soit le joueur est un joueur évolué, implémentant une heuristique de jeu (nous verrons plus tard ce que c'est), et étant capable de choisir le meilleur coup possible pour le jouer.

Dans cette sous partie, nous nous attacherons à décrire la manière que nos joueurs utilisent pour jouer un coup. Le choix du coup se faisant soit aléatoirement, soit par un algorithme de recherche que nous verrons plus tard, nous ne traiterons pas ceci dans cette sous partie.

Pour jouer un coup, le prédicat *joue_coup/4* est appelé. Il permet, pour une couleur, un plateau de jeu et un coup donnée, de récupérer un plateau de jeu (le quatrième argument du prédicat) ayant subit toutes les modifications dues au coup joué donné en paramètre.

Ce prédicat va alors faire appel à un prédicat *flipper/4* qui prend 4 arguments (un plateau de jeu, un coup *coup(I,J)*, une couleur, et une liste de pion quel renvoie). Ce prédicat va en fait déterminer tous les pions qui flippent lorsque le coup est joué. Ainsi, ce prédicat, à la manière des prédicats de vérification vus précédemment dans *position_est_jouable*, va parcourir toutes les directions. Dés qu'un pion doit changer de couleur, sa position est mémorisée dans une liste.

Lorsque toutes les vérifications sont faites, *flipper* renvoie une liste de pion à retourner. Puis, *joue_coup* va appeler *majPlateau/4* qui prend 4 arguments. On lui transmet alors un plateau de jeu, une couleur, et une liste de pions à retourner (typiquement, la liste retournée par *flipper*). Elle unifie alors un des arguments à un nouveau plateau ayant les pions de la liste placés aux bons endroits à la

bonne couleur, ainsi que les pions présents dans le plateau donné en paramètre à *joue_coup* et qui n'ont pas été retournés.

Enfin, *joue_coup* appelle une autre fonction auxiliaire qui va placer le pion que le joueur joue à la bonne case dans le plateau.

Il est à noter qu'à aucun moment dans l'algorithme il n'a été vérifié que la case courante est une variable (une case libre). En effet, il est supposé que le coup donné en paramètre est un coup légal. Et donc, que la case est bien libre et permet de prendre en sandwich des pions adverses. Dans la pratique, le coup a été choisi par *tous_coups_legaux,* que ce soit un joueur aléatoire, ou un joueur intelligent. On s'est alors assuré que le coup est valable.

3. Choisir un coup en mode aléatoire

Comme nous l'avons vu, un joueur aléatoire choisit un coup aléatoirement parmi une liste de coups qui lui sont permis dans une situation de jeu donnée. Ainsi, le joueur aléatoire fait appel au prédicat *choisi_coup/2* qui prend en premier argument une liste de coup, et renvoie un coup choisit aléatoirement dans cette liste via le second argument. En supposant que la liste de coup fournie provient du prédicat *tous_coup_legaux*, elle ne contient que des coups valides. Donc le coup choisit par *choisi_coup* est forcément valide.

4. Heuristique de jeu : prédicat d'évaluation

Afin de choisir un coup à jouer, il existe plusieurs stratégies possibles. La première est celle du joueur aléatoire, qui choisi un coup aléatoirement dans une liste de coup jouable pour un état donné. La seconde consiste à choisir un coup en utilisant une heuristique de jeu (une stratégie de jeu), à l'aide d'un algorithme de recherche que nous verrons plus tard dans ce document.

Le joueur aléatoire n'ayant pas besoin d'heuristique de jeu, nous ne parlerons pas davantage de lui dans cette partie du rapport.

Nous avons décidé de développer une heuristique de jeu afin que le programme joue les coups les plus intéressants selon nous. Une heuristique de jeu, c'est en fait une fonction qui va pouvoir évaluer un plateau de jeu, afin de savoir s'il est favorable, ou non, pour un joueur courant.

Notre toute première fonction d'évaluation était une fonction qui comptait simplement la différence de pion qu'un joueur avait avec le nombre de pion de son adversaire. Mais cette stratégie s'avérait rapidement très limitée. C'est pourquoi nous nous sommes instruits sur certaines stratégies efficaces en Othello. C'est ce que nous allons voir maintenant.

Nous avons alors implémenté une première stratégie, que nous appellerons stratégie simple, qui permet de mettre des valeurs à la position d'un pion sur le plateau. Les coins ayant la meilleure évaluation alors que les cases adjacentes à cette diagonale ont des évaluations négatives. Dans les

faits, nous distinguerons les pions « X », adjacent à la diagonale, et situé eux même sur la diagonale du plateau, et les pions « C », adjacent à la diagonales, mais situé contre un bord du plateau de jeu.

Cependant après une analyse des stratégies du jeu Othello, nous avons remarqué que cette stratégie n'était pas forcement optimale car elle ne prend pas en compte la stratégie dite des pions définitifs (pions qui ne peuvent être retournés par l'adversaire).

Nous avons alors amélioré la première heuristique en mettant de grande évaluations positives sur les pions adjacents aux coins si celui-ci nous appartenais déjà car ils ne pourront plus jamais être retourné par notre adversaire. C'est un excellent coup à jouer.

Nous avons ensuite implémenté la technique dite d'insertion. Cette technique permet d'insérer un pion sur les lignes 1 ou 6 ainsi que les colonnes A et F afin de pouvoir prendre un coin dans les coups futur.

Enfin, nous avons adapté la technique dite de mobilité qui limite le nombre de coup jouable de l'adversaire.

Notre stratégie finale est composée de trois phases. La première phase sert pour les deux premiers coups du jeu. Pour cet état nous avons choisi de limiter la mobilité de l'adversaire. Ensuite nous évaluons la valeur de la grille avec toutes les techniques décrite précédemment. La dernière phase, qui est composée des deux derniers coups, optimise le nombre de pions du joueur courant.

5. Algorithme de recherche dans un arbre de jeu : MinMax et Alpha-Beta.

Pour rechercher le meilleur coup qu'il faut faire pour battre un adversaire, il existe différent algorithmes de recherche permettant de déterminer avec une bonne précision (si l'heuristique est complète et efficace) le coup optimal pour une situation de jeu donnée.

Ainsi, nous verrons principalement deux algorithmes que nous avons employés pour nos joueurs Prolog utilisant des heuristiques de jeu : l'algorithme du MinMax et l'algorithme de l'alpha-beta. Ces algorithmes sont commentés en détail dans le code Prolog de chacun des joueurs heuristiques.

L'algorithme alpha-beta est une optimisation de l'algorithme du MinMax. L'algorithme MinMax est un algorithme de théorie des jeux à deux joueurs pour des jeux à sommes nulles. L'algorithme MinMax récupère alors pour le joueur qui à la main tous les coups qui lui sont possible de jouer. Pour chacun de ces coups, il le joue et relance l'algorithme du MinMax en décrémentant la profondeur et en donnant la main à l'autre joueur. Lorsque la profondeur atteint zéro, le MinMax évalue avec l'heuristique du joueur la situation de jeu courante et mémorise le coup qui a permis d'atteindre cette situation. Pour chacun des coups effectués, l'algorithme garde en mémoire successivement le meilleur coup (celui ayant l'évaluation la plus grande pour un joueur donné).

En fait, MinMax explore l'arbre de jeu jusqu'à une certaine profondeur donnée. Il fait remonter aux nœuds la meilleur évaluation (ou la moins bonne, ca dépend si on est à un niveau de maximisation ou de minimisation, respectivement, un niveau ami, ou ennemi). Pour chaque

évaluation, on connait le coup qui permet de l'atteindre. Ainsi, à la racine de l'arbre, on récupère le coup qui permet d'aller dans la branche ayant la meilleure évaluation.

L'alpha-beta est une amélioration de l'algorithme du MinMax dans le sens où toutes les feuilles ne sont pas nécessairement explorées. En effet, cet algorithme évite d'évaluer des nœuds de l'arbre de recherche dont on est sûr que leur qualité sera inférieure à un nœud déjà évalué. Ceci est fait grâce au prédicat *cutoff*.

Néanmons, il est à préciser que, sur nos PC personnels, MinMax fonctionne dans un temps raisonnable (profondeur 5/6) en une vingtaine/trentaine de secondes pour un coup. Pour l'alpha-beta, il faut entre 20 et 40 secondes pour un coup niveau 10/11. Les fourchettes de temps dépendent de la grille de jeu fournie à l'algorithme. Nous n'avons pas pu tester ces algorithmes interfacés avec l'arbitre sur les ordinateurs de l'Institut Galilée. Nos derniers tests remontent au Mercredi 16 décembre, nous montions en profondeur cinq pour MinMax, et sept pour alpha-beta. Mais les codes ont été optimisés depuis, et nous avions quelques bugs ce jour là. Il est ainsi possible que nous soyons amenés à modifier la profondeur de recherche d'ici le tournoi.

6. Améliorations apportées aux algorithmes au cours du codage

Au cours du codage, et dans un souci d'optimisation de l'utilisation du temps CPU, nous avons du modifier le code à certains endroits.

Ainsi, nous avons été amenés à rajouter des cuts aux endroits où nous sommes sûrs qu'un choix ne sera pas remis en cause dans le futur. Cela a permis d'éviter des backtracks inutiles, et notamment des boucles infinies. En effet, nous avons eu un souci lors de l'interfaçage de notre code avec celui de l'arbitre. Nous pouvions alors lancés l'algorithme de l'alpha-beta à une profondeur onze sans problème sur une grille de jeu, alors qu'une fois interfacé, avec l'arbitre, la profondeur 4 ne pouvait être dépassé. Nous avions sinon une erreur « Out of local stack », synonyme très souvent de boucle infinie. L'ajout de cut nous à permis de gagner un niveau de profondeur. Puis, l'ajout d'un seul cut, dans le code de l'alpha beta, nous a permis de doubler la profondeur, et en fait, de solutionner le problème de 'stack overflow'.

En outre, nous avons modifié l'ordre de définition de nos prédicats récursifs. Ainsi, nous nous sommes efforcés de mettre tous les cas d'arrêts en premiers, et les cas récursifs en derniers. Ceci à contribuer à améliorer la vitesse d'exécution de notre programme.

C. Tests des algorithmes principaux

Nous présentons dans le fichier 'othelloAB' notre travail global. Ainsi, c'est le fichier que nous avons commencé depuis la date de début du devoir. En fait, il représente selon nous la qualité de notre travail dans le sens où il est très commenté, dispose de nombreux exemples d'utilisations des différents prédicats, indique notre manière de procéder pour certains algorithmes et commente

certain des problèmes que nous avons rencontrés. Cependant il est à noter que ce fichier ne contient pas les dernières versions des algorithmes. Ainsi, ceux à prendre en compte sont ceux implémentés dans les joueurs (les toutes dernières versions étant celles des joueurs « alfabeta* », que nous verrons plus tard). Malgré cela, nous voulons inclure ce fichier à notre devoir car nous pensons qu'il constitue vraiment un appui quant à la démarche qualité de notre rendu.

III. Les joueurs

Les joueurs que nous allons présenter dans cette partie du rapport sont tous pleinement fonctionnels, et sans bug connu à ce jour. Nous avons essayé toutes les combinaisons de match possible entre ces joueurs. Aucun bug n'est intervenu ni aucun coup illégal. Tous ces joueurs sont disponibles pour l'arbitre (leurs chemins ont été ajoutés dans le fichier binome.pl). Les joueurs seront traités par ordre de numérotation à partir du fichier binome.pl. Ainsi, random1 et random2 sont disponibles sous les identifiants 1 et 2, minimax1 sous 3, alfabeta1 sous 4, etc.

Il est à noter qu'une brève description de chaque joueur est disponible dans le fichier texte 'Description Joueur', au début, situé dans le répertoire du jeu. En outre, des exemples d'exécutions de match sont présentés et analysés succinctement.

En outre, nous avons commenté le code des joueurs. Ainsi, certaines explications sur les algorithmes seront disponibles dans ces fichiers. En outre, des exemples de tests, ou des requêtes prêtes à être lancées, de nos prédicats sont présents dans ces fichiers.

A. Les joueurs random1 et random2

Ce sont nos deux premiers joueurs implémentés. Ils nous ont ainsi permis de tester rapidement que le mécanisme de gestion de coup (quels sont les coups légaux ? choisir un coup aléatoirement parmi cette liste de coups possibles) fonctionne. En outre, ils nous ont permis de faire fonctionner l'arbitre. Nous avions alors un joueur de base fonctionnel sur lequel nous appuyer afin d'ajouter les algorithmes de recherches tels que MinMax et alpha-beta.

Ces deux joueurs sont implémentés dans les répertoires *random1* et *random2* de l'archive que nous fournissons.

Leur principe de jeu est des plus simples. Lorsque c'est à leur tour de jouer, ils récupèrent le coup joué par l'adversaire à partir de l'arbitre, ils exécutent ce coup sur leur plateau de jeu local (sauf dans le cas où le joueur courant commence la partie, dans ce cas, il joue directement dans le plateau initial). Puis, ils calculent, via le prédicat vu précédemment *tous_coups_legaux* les coups possibles

pour le plateau de jeu courant. Ensuite, ils choisissent un coup aléatoirement via le prédicat *choisi_coup*. Ils jouent ce coup dans leur plateau local, puis envoie ce coup joué à l'arbitre. L'opération recommence jusqu'à ce que les deux joueurs passent consécutivement, ou qu'un joueur ne puisse jouer car le plateau est plein (il envoie alors à l'arbitre un coup « passe », car il ne peut jouer, puisque pas de coup disponible, l'arbitre coupant alors les deux threads de jeu).

Une partie random1 contre random2 se joue extrêmement rapidement. En moins d'une seconde sur nos machines personnelles (en soustrayant le temps de 10 secondes pendant lequel l'arbitre « dort » et le temps nécessaire à la sélection des joueurs).

Le joueur aléatoire que nous présentons au tournoi est, aléatoirement, random1 ou random2.

Dans les deux cas, le code étant identique, cela ne changera rien. Sauf si on commence à considérer qu'un des deux est plus « chanceux » que l'autre.

B. Les joueurs à heuristiques

1. Le joueur minimax1

Le joueur minimax1 a été le deuxième joueur que nous avons implémenté.

Directement basé sur le code des joueurs aléatoire, il rajoute à ces derniers notre heuristique intelligente ainsi que l'algorithme de recherche MinMax. Il permet alors de lancer des parties jusqu'en profondeur 6 (sur nos machines, il sera peut être nécessaire de réduire cette profondeur sur les ordinateurs de l'Institut Galilée à 5).

Implémenté dans le répertoire du même nom, minimax1 possède un assez bon taux de réussite contre des joueurs de types aléatoires. Mais ceci n'est encore rien par rapports aux joueurs, que nous verrons plus tard, qui sont, tout simplement, imbattables par un joueur aléatoire à partir d'une certaine profondeur de jeu.

Comme le code est récupéré à partir des joueurs random, il n'est pas nécessaire de décrire à nouveau le système de gestion de coups. En effet, le principe est le même pour tous les joueurs. La seule différence réside dans le fait que, pour les joueurs à heuristique, plutôt que de faire appel à *tous_coups_legaux* puis *choisi_coup*, on appelle directement l'algorithme de recherche concerné. Ici, c'est *minimax* qui est appelé.

Comme nous avons implémentés des algorithmes plus efficaces, nous parlerons dorénavant très peu de minimax1.

2. Le joueur alfabeta1

Le joueur alfabeta1, qui est notre joueur le plus évolué d'un point de vue algorithmique et heuristique, est implémenté à travers le fichier joueur présent dans le dossier alfabeta1.

Capable de battre tous les joueurs que nous avons implémentés, il possède notre heuristique dite intelligente. Lancé sur une profondeur de 10 (il sera peut être nécessaire de réduire cette profondeur, comme minimax1, à 7/8 sur les ordinateurs de l'école), il calcule à chaque fois qu'il a la main le meilleur coup possible en utilisant l'algorithme de l'alpha-beta. Il termine alors une partie en moins de 900 secondes (500-600 en moyenne).

Nous présentons dans le fichier 'Description Joueur' les matchs les plus intéressants selon nous d'alfabeta1.

Le joueur heuristique que nous présentons au tournoi est **alfabeta1**, car il représente selon nous notre meilleure chance de victoire.

3. Le joueur alfabeta2

Tout comme alfabeta1, alfabeta2 se base sur l'algorithme de recherche alpha-beta afin de déterminer le meilleur coup possible à chaque fois que c'est à son tour de jouer. En revanche, l'heuristique utilisée ici est celle que nous avons appelé précédemment l'heuristique « simple ».

Le code de ce joueur est disponible dans le fichier 'my_player.pl' situé dans le répertoire alfabeta2 du répertoire de jeu.

Le temps CPU pour faire une partie est, sur nos machines, d'environ 450 secondes (selon la tournure de la partie toutefois). Ce temps est plus court que celui d'alfabeta1. Ceci peut s'expliquer par le fait que la fonction d'évaluation est plus beaucoup plus simple, donc plus rapide à exécuter. En profondeur 10, le nombre de feuille à évaluer n'étant pas négligeable, ceci explique ce temps CPU raccourci.

4. Le joueur alfabetaSM

Ce joueur est identique au joueur alfabeta1, à la seule différence qu'il n'utilise pas le prédicat permettant de calculer la mobilité du joueur adverse. Ainsi, il utilise une sorte d'heuristique intelligente, mais du coup moins évoluée que celle d'alfabeta1.

Nous avons créé ce joueur dans le but de comparer l'apport de la stratégie de mobilité dans notre heuristique. A force de tests, il nous a permis de déterminer les valeurs des coefficients à attribuer à chaque stratégie implantée. Ceci nous a permis d'optimiser notre heuristique.

Implémenté dans le fichier de jeu présent dans le répertoire du même nom, alfabetaSM

Le temps CPU est ici d'environ 480 secondes. Plus cours que celui d'alfabeta1, ceci peut s'expliquer par l'absence de calcul de la mobilité. Toutefois, ce temps est plus long que celui d'alfabeta2 car le prédicat d'évaluation est plus complexe.

C. Tableau récapitulatif des victoires

Nous allons résumer ici les différentes victoires en moyennes de chacun des joueurs contre les autres, sur les joueurs les plus intéressants (minimax ne pouvant aller loin en profondeur, il se fait très souvent battre par les alpha-beta, mais il bat souvent random1/2). Ce tableau se base sur les matchs effectués dans le fichier 'Description Joueur', mais aussi sur nos tests que nous avons effectués tout au long de l'implémentation. Nous considérerons, pour les algorithmes utilisant alpha-beta, une profondeur égale pour tous les joueurs, à savoir 10.

Légende :

- A bat B : **V**
- A perd contre B : **P**
- Il est impossible de déterminer qui gagne le plus en moyenne : **X**

A \ B	random1	random2	alfabeta1	alfabeta2	alfabetaSM
random1	-	X	P	P	P
random2	X	-	P	P	P
alfabeta1	V	V	-	V	V
alfabeta2	V	V	P	-	V
alfabetaSM	V	V	P	P	-

Ce tableau nous apporte un enseignement important : alfabeta2 l'emporte, contre toute attente contre alfabetaSM.

En fait, on peut déduire simplement le constat de cette partie riche d'enseignement.

En Othello, un joueur qui commence en deuxième position est avantagé par rapport au joueur qui débute la partie en noir, selon nos recherches sur les stratégies. En outre, il est aussi possible que l'appui de la stratégie de mobilité manque ici pour qu'alfabetaSM l'emporte. Il faudrait alors optimiser les coefficients de notre fonction d'évaluation afin d'être certain que notre alfabetaSM, (et en fait, notre alfabeta1 d'une manière indirecte car les fonctions d'évaluation sont presque similaires) puisse battre une intelligence de jeu simple dans tous les cas.

Lorsqu'on pousse notre série statistique plus loin, on se rend en fait compte que notre heuristique simple rivalise, et peut même battre, une heuristique évoluée comme celle d'alfabeta1. Il lui faut pour cela commencé en deuxième, et la victoire peut être au bout.

IV. Améliorations

Au cours de l'implémentation de nos joueurs d'Othello, nous nous sommes rendu compte qu'il était possible d'effectuer certaines améliorations. Par manque de temps, nous ne pouvons les implémenter, mais leur intérêt nécessite toutefois que nous en parlions.

Une des contraintes les plus importantes à prendre en compte pour ce projet est celle du temps de calcul, qui est limité à 900 secondes de temps CPU. Le problème majeur vient du fait que la création de l'arbre de jeu peut se révéler très longue. Ainsi, l'apport de l'algorithme de l'alpha-beta par rapport au minimax nous a permis de pratiquement doubler la profondeur de recherche (nous sommes passés d'une profondeur de cinq/six sous MinMax, à environ onze pour l'alpha-beta).

Toutefois, nous pouvons encore réduire cette contrainte de temps en utilisant d'autres algorithmes tels que l'algorithme du négascout qui permet de décider de la profondeur de recherche d'une branche selon certains critères. Ainsi, une branche qui semble bénéfique pour le joueur sera explorée plus profondément qu'une branche qui semble être moins bonne. Un tel algorithme permet de gagner en moyenne 10% de temps comparé à l'utilisation de l'alpha-beta seul. On pourrait alors aller plus en profondeur pour explorer les feuilles de l'arbre de jeu, et ainsi faire remonter un meilleur coup.

Il est aussi possible d'utiliser un alpha-beta à mémoire de coup car nous effectuons successivement plusieurs appels identiques. En effet, supposons que nous ayons un alpha-beta qui recherche à une profondeur de dix. On posera alors le pion, à la position retournée par l'algorithme, qui nous amènera à la situation la plus favorable dans les dix coups. Or lors du prochain coup, nous effectuerons une recherche en profondeur qui contiendra un sous arbre correspondant exactement à un sous arbre déjà effectué précédemment, avec une profondeur inferieur de deux coups néanmoins. Avec un algorithme à mémoire de coup, on économise le temps de calcul de ce sous-arbre.

Ces techniques peuvent à nouveau être améliorées en ayant un meilleur tri des coups. Le cas le plus rapide étant celui ou on explore le meilleur nœud en premier.

Une autre piste d'amélioration est l'heuristique du système de jeu. Il serait bénéfique d'implémenter une bibliothèque d'ouverture afin d'avoir le meilleur début de partie possible. Cependant le temps imparti est insuffisant pour l'implémentation de plusieurs centaines de cas possibles.

Il serait aussi envisageable d'implémenter d'autres techniques et stratégies de jeu tel que le bétonnage, les coups d'attente, la technique du bord de cinq ou encore le piège de Stoner pour le milieu de partie, afin d'optimiser le taux de réussite de nos joueurs à heuristiques.