Ecole d'ingénieurs Sup Galilée
Université Paris 13
2009-2010

Programmation Logique

Othello 6x6

FABRE Morgan

GIRARD Yohann

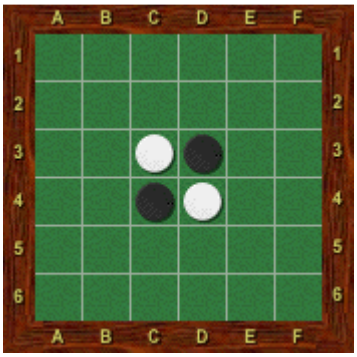
Sommaire

Introduction

Le but de ce devoir de programmation logique était de concevoir un programme PROLOG permettant de jouer au jeu d'Othello. Il s'agit d'un jeu à deux joueurs qui se joue au tour à tour sur un damier de 8x8 cases en général. Cependant, il peut être joué sur des damiers plus petits ou plus grands. Dans le cadre du devoir, le damier aura une taille de 6x6.

Rappel des règles :

- Le joueur qui joue en premier est le joueur possédant les pions noirs.
- Est considéré comme légal, tout coup prenant en sandwich au moins un pion ennemi sur un axe horizontal, vertical ou (non exclusif) diagonal.
- Lorsqu'un joueur pose un pion sur le damier, tous les pions ennemis pris en sandwich sont retournés et leur couleur est alors celle du joueur venant de jouer.
- Si un joueur ne possède pas de coups légaux, il passe et c'est au tour du joueur adverse.
- Si un joueur possède au moins un coup légal, il doit jouer.
- Si les deux joueurs passent successivement leur tour, la partie est finie.
- Le gagnant est celui possédant le plus de pions sur le plateau au terme de la partie.
- Le plateau de jeu est initialement comme suit :



L'énoncé du devoir ne décrivait pas le damier de départ ainsi, mais l'arbitre le décrit de cette manière-ci. L'arbitre a toujours raison.

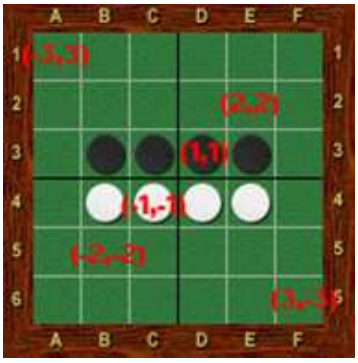
Structures de données

Lorsque nous avons commencé notre programme, nous ignorions que PROLOG autorise la définition de couples. Nous avons donc défini des couples comme étant des listes à deux éléments. En outre, tout ce que nous appellerons couple dans la description de nos structures de données sera modélisé par une liste.

Plateau de jeu

Nous avons décidé de représenter un plateau de jeu comme étant un couple (PN, PB) avec PN la liste des pions possédés par le joueur noir et PB la liste des pions possédés par le joueur blanc. Ce choix a été motivé par notre implémentation de la récupération des coups légaux. En effet, nous avons constaté que pour qu'un coup soit légal, il doit obligatoirement être adjacent à une case ennemie. Par conséquent, pourquoi parcourir toutes les cases du damier pour trouver un coup légal alors qu'il suffit seulement de parcourir la liste des pions ennemis.

Pour représenter une case, nous utilisons un couple de coordonnées. Cependant, ces coordonnées sont assez spéciales car avant de commencer notre programme, nous avions déjà réfléchi à plusieurs aspects du jeu et notamment à la gestion des symétries. Il n'est pas simple de démontrer qu'un plateau est ou n'est pas symétrique avec des cases numérotées de 1 à 6 alors que cela est relativement simple si l'on sépare le damier en son milieu.



Le damier de gauche est représenté ainsi :
[[[-2, 1], [-1, 1], [1, 1], [2, 1]], [[-2, -1], [-1, -1], [1, -1], [2, -1]]]
On voit bien sur l'image qu'il est symétrique verticalement.
Pour le prouver grâce aux coordonnées des cases, il suffit de regarder que chaque pion noir dans la partie verticale gauche du damier possède son équivalent dans la partie verticale droite. Même chose pour les pions blancs.
Noir : [-2, 1] -> [2, 1] ; [-1, 1] -> [1, 1] ; OK
Blanc : [-2, -1] -> [2, -1] ; [-1, 1] -> [1, -1] ; OK

En résumé, un damier est donc représenté par la liste des cases non vides qu'il contient, ces cases étant séparées suivant leur propriétaire.

Coup légal

Un coup légal est une case vide qui permet à un joueur de retourner un ou plusieurs pions adverses. Par rapport à notre structure de jeu, un coup légal est représenté comme un couple (Case, Liste) où Case est le couple de coordonnées de la case où le pion du joueur va être placé et Liste est la liste des cases ennemies qui seront retournées.

Joueur aléatoire

Le joueur aléatoire représente le plus simplet des joueurs qui soient. Il se contente de récupérer la liste des coups légaux d'un plateau de jeu et en sélectionne un au hasard pour le jouer.

Recherche des coups légaux

La construction de la liste des coups légaux se fait grâce à la fonction *tous_coups_legaux(+Couleur, +Etat, -ListeCoupsAdmissibles)*. Cette fonction récupère la liste des pions adverses et appelle une sous-fonction *tous_coups_legaux1/7* qui va traiter récursivement cette liste de pions. Pour chaque pion adverse, on construit alors la liste des coups légaux qui lui correspondent en parcourant les cases vides adjacentes grâce à *coups_legaux/7*. Il est possible que pour plusieurs pions différents, on trouve un même coup légal. Par conséquent, une liste des cases explorées est donnée en argument à la fonction de recherche des coups légaux. Cette liste est d'ailleurs mise à jour et renvoyée pour un futur appel. Ainsi, une case n'est explorée qu'une seule et unique fois.

Pour déterminer si une case est un coup légal ou non, on appelle la méthode *jetons_retournes/6*. Cette méthode construit la liste des pions qui seront retournés si le joueur joue ce coup. Si la liste est vide ce n'est pas un coup légal, sinon c'en est un.

La méthode *jetons_retournes/6* vérifie tout d'abord qu'il s'agit d'une case vide. Si tel est le cas, elle regarde autour d'elle dans toutes les directions. Pour chaque direction, si la case adjacente est une case ennemie, alors elle appelle la fonction *sandwich/6* qui va construire la liste des pions retournés dans une certaine direction. La liste finale est donc la concaténation des listes obtenues en parcourant toutes les directions.

La méthode *sandwich/6* accumule dans une liste toutes les cases qu'elle reçoit tant que ces cases sont des cases ennemies. Si elle tombe sur une case vide, alors elle échoue. Si elle tombe sur une case ennemie, elle se rappelle récursivement avec la case adjacente dans la direction observée. Sinon, c'est une case amie et elle renvoie la liste des cases accumulées.

Sélection aléatoire d'un coup

La sélection aléatoire d'un coup est triviale, elle se contente de récupérer la liste des coups légaux grâce à la méthode décrite précédemment puis elle fait un random entre 0 et NbCoups-1 pour choisir un coup dans la liste.

Jouer un coup

Jouer un coup est également assez simple puisqu'il suffit d'ajouter dans la liste des pions du joueur le pion à jouer et la liste des pions retournés puis, de retirer la liste des pions retournés dans la liste des pions du joueur adverse. Cette simplicité est due à l'efficacité de notre implémentation de *tous_coups_legaux/3*.

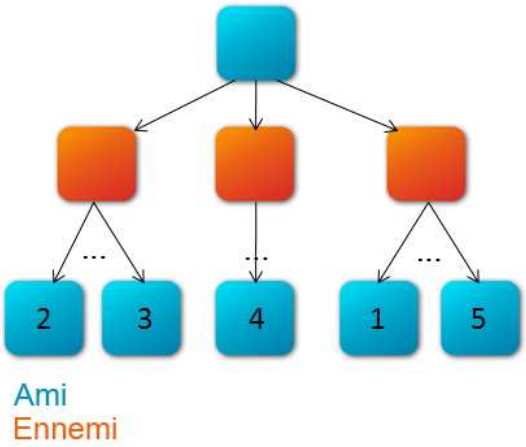
Bonus

Du fait de nos structures de données permettant de représenter un damier comme une liste de cases, le programme PROLOG que nous avons développé est adaptable à tous les damiers. En effet, il permet de gérer des damiers de tailles autres que 6x6. A titre d'exemple, nous avons créé un fichier aleatoire.pl contenant une fonction *lancer_partie/0* qui demande à l'utilisateur le nombre de lignes/colonnes du damier et lance une partie entre deux joueurs aléatoires sur un damier de la taille désirée.

Pour une question de performances, cette adaptabilité n'a pas été gérée pour le joueur intelligent. Pour comprendre cela, il suffit de regarder le code source au niveau du prédicat *voisin_superieur/2*. Cependant, adapter le joueur intelligent pour un plateau de taille non fixée serait relativement simple, même si ce ne serait pas d'une grande utilité du fait de l'heuristique choisie. La taille du damier est une chose très importante à prendre en compte lors de l'élaboration d'une heuristique.

Joueur intelligent

Le joueur intelligent est un joueur qui va évaluer quel est le meilleur coup selon lui parmi la liste des coups admissibles. Pour ce faire, il va pour chaque coup, observer ce que l'adversaire peut faire. A partir de ce que l'adversaire peut faire, il va regarder ce que lui-même peut faire… et ainsi de suite. Il va donc anticiper ses propres coups et ceux de l'adversaire. On peut schématiser cela par le graphe suivant :



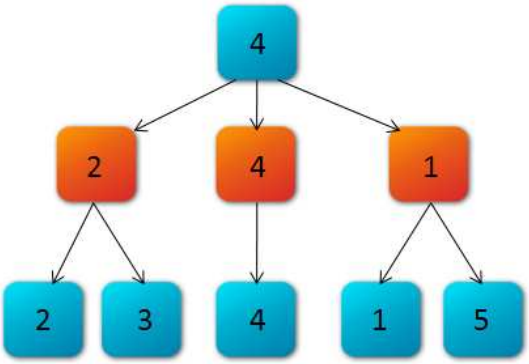
Dans le cas du jeu d'Othello, on ne peut jamais revenir sur un plateau antérieur, le graphe de jeu est donc un arbre. On peut donc lui associer une profondeur qui va alors correspondre au « raffinement » de l'anticipation des coups. Plus on descend dans l'arbre, plus on anticipe les coups d'un état du jeu.

Recherche du meilleur coup

Principe

L'objectif premier du joueur dit intelligent est de calculer quel est le meilleur coup à jouer pour lui. Pour ce faire, il doit créer une stratégie de jeu, appelée heuristique. Cette heuristique va permettre d'évaluer la valeur d'un plateau en lui attribuant une note. Plus la note est élevée, plus le plateau lui est favorable. On pourrait se dire qu'il suffit de descendre dans l'arbre et de récupérer la valeur maximale du plateau au niveau ami (ou valeur minimale du plateau au niveau ennemi) et de jouer le coup qui a amené ce plateau. Mais faire cela, serait sous-estimer l'adversaire. En effet, il faut supposer que l'adversaire aussi est intelligent et qu'il anticipe également nos coups. Il se peut que dans la branche qui mène au plateau de note maximale, il existe un nœud possédant un état fils qui nous soit très défavorable. Dans ce cas, l'adversaire va jouer de manière à obtenir cet état et on n'atteindra jamais l'état que nous souhaitions atteindre. C'est le cas de la feuille de valeur 5. Cette feuille serait un bon état pour le joueur courant, mais si le joueur adverse joue bien, il ne lui permettra pas de l'atteindre et jouera de façon à obtenir la feuille de valeur 1.

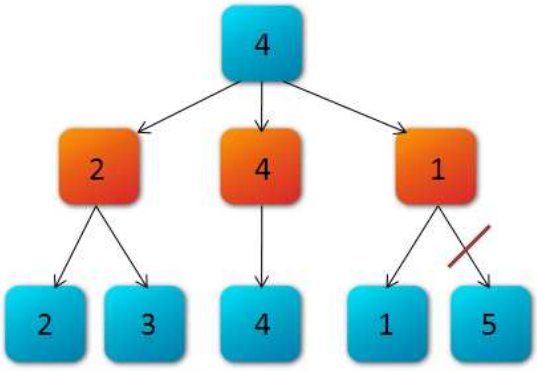
Par conséquent, il faut donc faire remonter la valeur qui maximise notre coup et minimise celui de l'ennemi. C'est le principe du MiniMax.



Le meilleur coup à jouer est donc le deuxième d'après MiniMax.

L'algorithme du MiniMax est efficace pour obtenir ce que l'on recherche, mais il a le défaut d'être lent. Il suffit de regarder l'arbre de parcours pour voir que beaucoup d'états vont être générés. Plus la profondeur est grande, plus l'algorithme est lent.

Pour contrer ce problème, il existe un algorithme nommé Alpha-Beta qui effectue exactement la même chose que le MiniMax mais d'une manière plus rusée. En effet, Alpha-Beta part du principe que, si l'on recherche le minimum à un certain étage, l'étage supérieur va alors rechercher le maximum. Donc si la valeur observée est inférieure au minimum courant, ce n'est pas la peine d'aller plus loin puisque de toute façon, la valeur va décroître et on ne la prendra jamais puisque l'étage supérieur recherche le maximum. Même chose pour les étages max qui possèdent un étage supérieur min. Alpha-Beta effectuant un parcours en profondeur d'abord, il peut permettre d'élaguer de très grosses parties de l'arbre.



1 étant inférieur à 4, on ne continue pas le parcours et 5 n'est jamais évalué.

Implémentation Alpha-Beta

L'implémentation d'Alpha-Beta nous a été donnée durant le devoir. Nous l'avons dans un premier temps adaptée à nos structures de données puis nous l'avons complétée pour les cas où il n'y a pas de coups admissibles à partir d'un état de jeu. En effet, il est possible que dans le parcours en profondeur, un joueur tombe sur un état où il ne peut pas jouer. Il existe alors deux possibilités. Soit le joueur doit passer et son adversaire va pouvoir jouer. Ce cas n'est pas favorable car cela permet à l'adversaire de prendre un coup d'avance. L'évaluation de ce coup est alors négative. Soit le joueur doit passer et son adversaire aussi. Il s'agit d'un double-passe qui implique une fin de partie selon les règles. On regarde donc qui est le gagnant et on renvoie une évaluation en conséquence.

Afin d'améliorer les performances d'Alpha-Beta, nous avons effectué un tri des coups admissibles du plus favorable, à priori, au moins favorable avant de faire le parcours. Ainsi, la probabilité que la recherche soit coupée tôt dans l'arbre est beaucoup plus importante puisque le meilleur coup a plus de chances d'être en début de parcours.

Le tri des coups est effectué à l'aide du prédicat *predsort/3*, qui fait partie de swi-prolog. Ce prédicat effectue un tri fusion suivant une liste et un prédicat qui détermine le signe d'ordre entre deux éléments (< ou >). Nous avons défini l'importance des coups de la façon suivante : coin > bord > case centrale > case C > case X

Les notions de case X et case C seront expliquées dans l'heuristique.

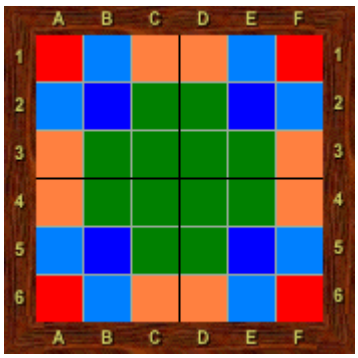
Dans le but d'améliorer encore plus notre Alpha-Beta, nous avons tenté deux solutions qui se sont avérées infructueuses. Tout d'abord, nous avons remarqué qu'Alpha-Beta effectue la recherche des coups légaux plusieurs fois pour un même plateau lorsque l'on avance dans la partie. Nous avons donc sauvegardé la liste des coups légaux des plateaux grâce à *asserta/0* et une fonction dynamique. Malheureusement, cette action a effectivement diminué le nombre d'inférences lors de l'exécution d'Alpha-Beta mais le temps de calcul lui, a largement augmenté. Ensuite, nous voulions réduire l'arbre de recherche en recherchant les symétries d'un état de jeu. Nous avions commencé l'implémentation de la recherche des symétries mais nous l'avons abandonnée en constatant que dans le jeu d'Othello, les symétries sont très rares et faire une recherche de symétries pour chaque plateau est finalement plus coûteuse en temps de calcul qu'autre chose.

Finalement, pour rendre notre recherche du meilleur coup plus pertinente, nous avons défini la profondeur en fonction du nombre de coups joués. Ainsi, lors du début de partie, nous effectuons une recherche en profondeur 8 et à partir du milieu de partie, vers le 19^{e}

coup, la profondeur passe à 10. Nous aurions pu mettre une profondeur 12 car dans nos tests, le temps maximal de 15 minutes n'a jamais été dépassé. Mais nous préférons jouer la prudence étant donné que les ordinateurs de l'université sont moins puissants et que suivant les coups joués par l'adversaire, le temps de parcours d'Alpha-Beta peut varier du simple au double, voire au triple. Donc plutôt que de risquer de prendre trop de temps et de devoir finir en sélection de coup aléatoire, nous préférons aller moins loin mais être sûrs de rester dans le temps imparti.

Heuristique

Sachant que le but du jeu est d'avoir le plus de pions et que pour manger des pions adverses, il faut les prendre en sandwich, on voit bien que les pions au centre du damier seront fréquemment retournés car ils sont vulnérables dans toutes les directions. De la même manière, on remarque que les cases en bordures du damier n'ont que deux directions dans lesquelles elles sont vulnérables et que les coins ne sont pas mangeables. Les coins sont donc des cases stratégiques qui peuvent tout changer selon la disposition du jeu. Les seules façons d'obtenir un coin sont de prendre en sandwich des pions en case C ou en case X. Ci-dessous, un schéma explicatif de la répartition des cases sur le damier d'Othello.



Coin
Bord
Case centrale
Case C
Case X

En général, les débutants jouent le coup qui va retourner le plus de pions adverses. Cette méthode de jeu est loin d'être efficace car tout pion retourné sera potentiellement retourné par l'adversaire au coup suivant.

Une autre heuristique utilisée par les débutants consiste à assigner une valeur pour chaque type de case. Ainsi, si un coup possible est un coin, alors on joue ce coup. Sinon, s'il n'y a pas de coin, on prend un bord. Et ainsi de suite. Le problème de cette heuristique est qu'elle ne tient pas compte de l'état du plateau. En effet, il peut être plus favorable de ne pas prendre le coin tout de suite et de le prendre plus tard parce que l'adversaire ne pourra pas le prendre, voire même de le laisser à l'adversaire à partir du moment où derrière on est sûr de gagner la partie.

Dans notre heuristique, nous avons décidé de prendre en compte les types de cases du damier car cela permet de juger notre potentiel offensif. En effet, un plateau où l'on possède les quatre coins a de grandes chances d'être meilleur qu'un plateau où l'on ne possède aucun coin car les quatre coins sont autant de pions définitifs et ils permettront sûrement de manger plus de pions adverses en fin de partie.

Nous avons donc attribué des points aux cases de la façon suivante :

- Un coin vaut 800.
- Un bord vaut 300.
- Une case centrale vaut 100.
- Une case C vaut la valeur d'un bord si le coin adjacent est occupé par le joueur, -400 sinon.
- Une case X vaut -500.

Une partie de l'évaluation d'un plateau est donc la somme des valeurs des pions du joueur ami – la somme des valeurs des pions du joueur ennemi.

Comme les cases possédées ne font pas tout, nous avons également pris en compte la mobilité. La mobilité d'un joueur est le fait qu'il possède plus ou moins de coups admissibles pour un état du jeu. Plus la mobilité d'un joueur est grande, mieux c'est pour lui car il a moins de chances d'être obligé de jouer un mauvais coup. On voit bien que restreindre la mobilité du joueur adverse est donc une bonne stratégie car cela peut le forcer à jouer des coups qu'il aurait préféré éviter. Afin de calculer la mobilité d'un joueur, on compte le nombre de pions adverses qui sont en frontière, la frontière d'un joueur étant l'ensemble de ses pions adjacents à une case vide.

Notre stratégie se résume donc à prendre en compte la position des pions sur le damier et la mobilité des deux joueurs, éviter au joueur de passer quand l'adversaire peut jouer derrière et à rechercher un état final tel que le joueur possède plus de pions que son adversaire. Pour parvenir à ce résultat, la profondeur d'exploration change en cours de partie pour observer plus loin lorsque l'on se rapproche de la fin.

Programme arbitre

Le programme qui arbitre une partie entre deux joueurs nous a été donné pour pouvoir adapter nos codes en vue du tournoi. Nous devions insérer les appels à nos fonctions dans des prédicats préconçus et créer deux prédicats « traducteurs ». Le premier prédicat permet de transformer un plateau au format de l'arbitre en un plateau à notre format et le second transforme une case dans notre format en une case au format de l'arbitre, pour pouvoir indiquer les coups à jouer à l'arbitre.

Pour tester notre programme à partir d'un plateau spécifique au format de l'arbitre, il suffit de faire ceci :

plateau_arbitre_prog(PlateauArbitre, NPlat), time(meilleur_coup(1, noir, NPlat, Coup)).

Ceci va convertir le plateau au format de l'arbitre en un plateau à notre format puis rechercher le meilleur coup de ce plateau pour le joueur noir, en considérant qu'il s'agit du premier coup de la partie.

Tests

Les tests qui figurent dans cette partie ne représentent pas l'ensemble des tests effectués. Faire figurer tous les tests que nous avons faits serait beaucoup trop long, sachant que ce rapport est limité à 15 pages.

Joueur aléatoire

Les tests sur le joueur aléatoire ont consisté à vérifier que la liste des coups légaux est exhaustive et que tous les coups calculés sont bien admissibles ainsi qu'à vérifier le bon fonctionnement du prédicat qui permet de jouer un coup et retourner des pions.

Coups admissibles

La liste des coups légaux pour le damier initial est correcte. Du fait de sa simplicité, nous allons prendre un cas plus complexe, où un coup va prendre en sandwich des pions ennemis dans plusieurs directions.

A partir de ce damier de milieu de partie, voici la liste des coups admissibles pour le joueur noir, avec les pions qu'il va retourner :

```
?- afficher_plateau(P), tous_coups_legaux(noir, P, CoupsLegaux).
 o x _ o _ o
 o o x o o o
 o x o x o o
 x x x o x o
 _ _ o o o o
 o o o o o o

CoupsLegaux = [[[2, 3], [[-1, 1], [1, 2], [2, 1], [2, 2]]]] .
```

Le joueur noir n'a donc qu'un seul coup admissible qui est E1. En jouant ce coup, il va retourner les pions adverses situés en C3, D2, E3 et E2.

Jouer un coup

Voici l'état du plateau après avoir fait jouer un coup à noir de manière aléatoire. Le coup choisi était prévisible puisque de toute façon il n'en avait qu'un.

```
o x _ o x o
o o x x x o
o x x x x o
x x x o x o
_ _ o o o o
o o o o o o
```

Othello géant

Comme nous l'avons mentionné, notre implémentation permet de créer des parties avec des damiers de tailles différentes. Voici un exemple de partie d'Othello 16x16.

```
?- lancer_partie.
Veuillez indiquer le nombre de lignes svp: 16.
```

……

```
 o o o o o o o o o o o o o o o o
 o x x o o x x x x x o o o o o o
 o x o o o x x x x o x o o o o o
 o o o o x x o x o x x o o x x o
 o o x x x x x o x o x o o x x o
 o o x o o x o x o o x o o o x o
 o o o x x x x x x x x x o o o o
 x o o o o o o x x o x x o o o o
 x o o x x x o o x o x o o o o o
 x o x o o o o o x o x x o o o o
 x o o x o o o o x x x x o o x o
 x o x x o x o o x x x x o x o o
 x o x x o x o x o x x x o o x o
 x x o x x o o x o o x x o o o o
 x x x x o o x o x x x x o o o o
 x x x o o o o o o o o o o o o o

FIN DE LA PARTIE d=')
Noir a 103 pions et Blanc en a 153.
Blanc est le grand gagnant! Il est trop fort à Othello :P
```

Joueur intelligent

Alpha-Beta

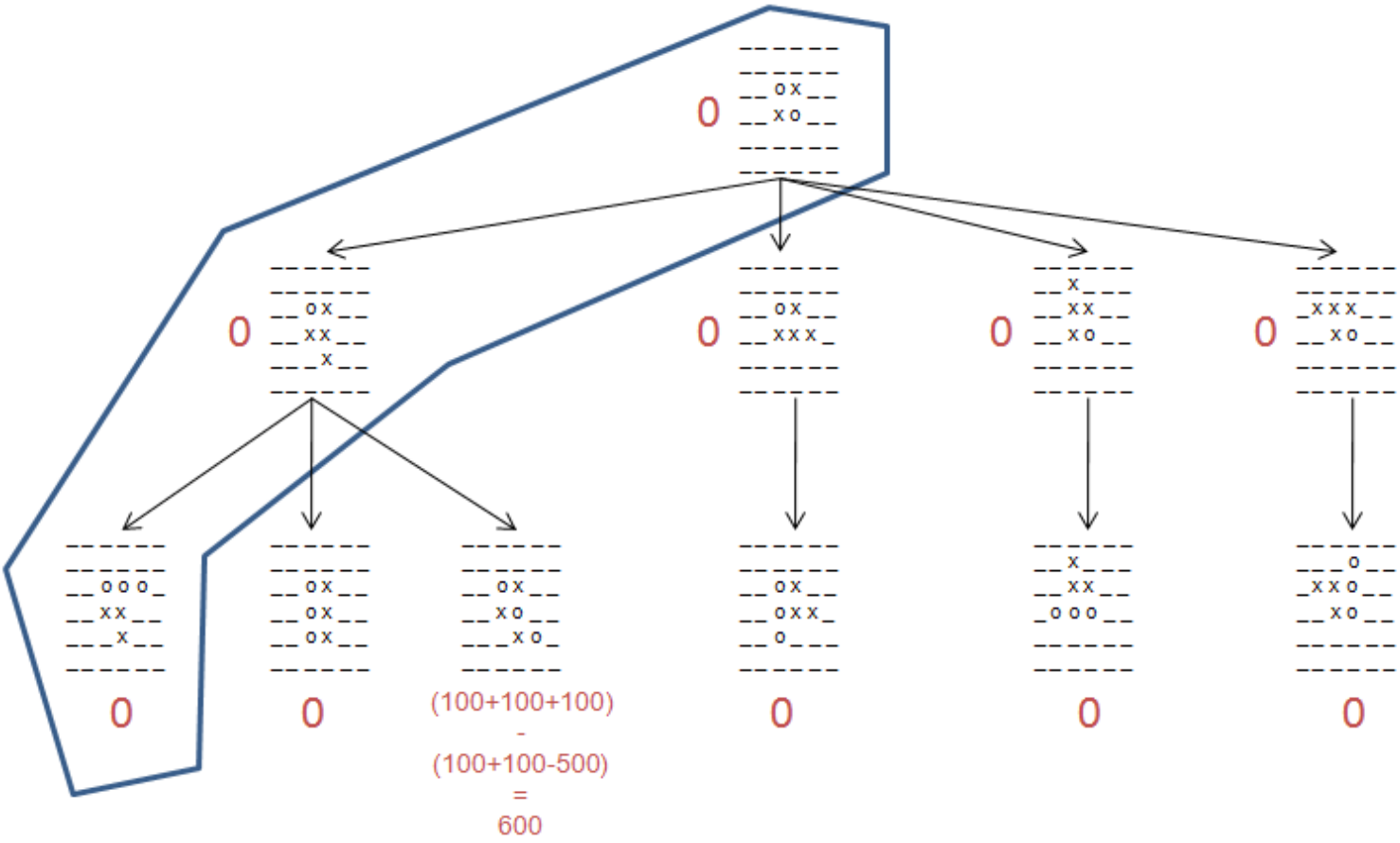
Voici le résultat d'Alpha-Beta en profondeur 2 sur le damier initial. C'est noir qui joue, comme le veut la règle du jeu.

?- init_plateau(P), alpha_beta(2, 2, noir, P, -9999999, 9999999, Coup, Valeur).

P = [[[1, 1], [-1, -1]], [[-1, 1], [1, -1]]],

Coup = [[1, -2], [[1, -1]]],

Valeur = 0.



L'arbre ci-dessus a été créé à partir d'affichages de plateaux dans le prédicat *evaluate_and_choose/9*. L'arbre de recherche a correctement été élagué et le résultat est celui attendu.

Heuristique

Afin d'améliorer notre heuristique, nous avons évidemment réfléchi sur des stratégies et nous sommes documentés sur Internet. Nous avons trouvé une multitude de sites intéressants qui expliquent diverses stratégies de jeu mais également différentes implémentations de la recherche du meilleur coup : NegaMax, NegaScout/PVS, MTD(F), … Nous avons tenté d'implémenter le MTD(F) couplé à Alpha-Beta avec mémoire mais n'avons pas réellement compris le principe de la mémorisation d'Alpha-Beta. De plus, vu l'effet qu'a produit la sauvegarde des coups légaux, nous étions sceptique quant à l'efficacité d'un Alpha-Beta avec mémoire. Nous avons donc gardé l'algorithme d'Alpha-Beta classique et nous sommes concentrés sur une stratégie de jeu.

Pour obtenir notre heuristique finale, nous avons implémenté plusieurs stratégies de jeu différentes et avons organisé une sorte de mini-tournoi. Nous avons par exemple créé une heuristique qui jouait un coin dès qu'elle le pouvait, une heuristique qui comptait le nombre de pions vers la fin de la partie, … Au final, notre heuristique était la meilleure.

Cependant, une chose qui nous a troublé, est qu'en augmentant notre profondeur, certaines heuristiques que nous battions aussi bien en tant que noir que blanc, nous battaient. Nous pensions qu'Alpha-Beta ne fonctionnait pas correctement, mais non, cela est dû au fait qu'il peut arriver qu'en parcourant l'arbre des coups possibles on s'arrête sur une feuille qui paraît être un bon coup alors que le coup suivant est désastreux. C'est pourquoi, il existe une partie de chance lors du choix de la profondeur. En effet, suivant l'heuristique du joueur adverse, aller un tout petit peu plus loin dans l'arbre peut s'avérer moins efficace, voire dramatique.

Preuve de son efficacité, notre heuristique est capable de battre le joueur aléatoire plus d'une fois sur deux. Pour dire vrai, le joueur aléatoire n'a même jamais réussi à nous battre alors qu'il est arrivé que nous lui fassions des parties parfaites. Nous avons d'ailleurs remarqué que c'est grâce à la restriction de la mobilité adverse et à l'augmentation de la nôtre que nous avons pu réaliser de telles parties. En effet, le joueur aléatoire n'avait plus que des solutions extrêmement désavantageuses pour lui, voire aucune solution. Ce qui nous permettait de manger des pions sans qu'il n'en mange.

En moyenne, nous avons calculé que la partie de notre joueur intelligent dure 300 secondes. Cependant, il arrive que la partie dure plus longtemps, jusqu'à 500 secondes. En mettant une profondeur de 12 en milieu de partie à la place d'une profondeur de 10, la partie dure en moyenne 750 secondes. Ce temps fait partie de l'intervalle règlementaire, mais comme dit précédemment, nous préférons jouer la carte de la sécurité et mettre une profondeur moins grande. Notre joueur est donc bien plus fort, mais il est également bien plus long que le joueur aléatoire qui est instantanée. Cependant, en mettant une profondeur de 6 durant toute la partie, avec une moyenne de 15 secondes par partie, le joueur intelligent n'a jamais laissé une seule partie au joueur aléatoire.

Conclusion

Notre gestion d'un jeu d'Othello en PROLOG fonctionne correctement pour un joueur aléatoire et un joueur intelligent. Le joueur intelligent que nous avons créé est effectivement meilleur qu'un joueur aléatoire, ce qui est le minimum requis pour qualifier un joueur d'intelligent.

Pour ce qui est du code généré, nous avons pris un soin particulier quant à sa lisibilité. Nous avons détaillé dans le rapport le principal des prédicats mais du fait de la limitation du nombre de pages, certains n'ont été que partiellement détaillés. Pour en savoir plus, vous pouvez lire le code source qui est clairement documenté avec pour chaque prédicat, des explications sur son utilité et son fonctionnement.

Pour finir, notre code est clairement modulaire et adaptable. La possibilité de choisir la taille du damier en est le plus bel exemple car cela permet de jouer au jeu d'une manière différente, sans devoir apporter la moindre modification.