 **Rapport TP - Algo Minmax, application au TicTacToe**

**NOM DU DOSSIER**

**Sous titre**

Date

Service

Balayssac Antoine, Rassat Théo

Semestre 2 - Année 2018/2019

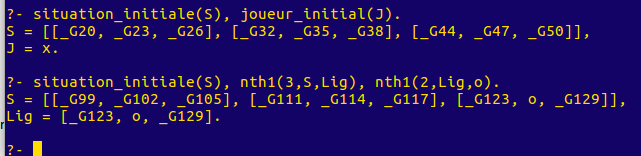
**Introduction :**

Au cours de ce TP, nous avons implémenté l’algorithme Minmax pour la convention Negamax. Le but est de rechercher le coup optimal dans un arbre de jeu à 2 joueurs depuis la situation initiale, pour une profondeur variant de 1 à 7 coups.

Nous suivrons au cours de ce compte-rendu de TP les étapes suivies lors de l’implémentation, en répondant aux questions posées, en y ajoutant éventuellement un screenshot de code. Dans ce même mail, sont fournis nos codes sources.

1. **Familiarisation avec le problème du TicTacToe 3x3**

**A - Quelle interprétation donnez-vous aux requêtes suivantes?**



* La première requête nous permet de récupérer la situation initiale dans la variable S. Elle est alors représentée par une matrice de variables non liées. Ce qui est possible dans ce jeu car au TicTacToe on doit instancier la grille. Les joueurs pourront ensuite lier ces variables par leur signe respectif (x,o). On ne crée ainsi pas de nouvelle grille à chaque modification.

Le joueur qui commence est récupéré par la seconde partie de la requête dans la variable J. Cette convention arbitraire est donnée dans le fichier tictactoe.pl. Les joueurs sont matérialisés par le signe “x” ou “o”. C’est le joueur x qui commence.

* La seconde requête récupère la situation initiale dans la variable S comme précédemment. Ensuite, grâce au prédicat Prolog nth1, on récupère premièrement la 3ème ligne de la matrice S dans la variable Lig et on lie ensuite le 2ème élément de celle-ci avec “o”. (nth1 commence à “compter” à 1). Cela modélise donc le fait que le joueur o a joué sur la situation initiale en 3x2.

**B - Alignements**

Il s’agit ici, de définir un prédicat alignement qui pour une matrice donnée, renvoie tous ses alignements possibles : lignes, colonnes et diagonales.

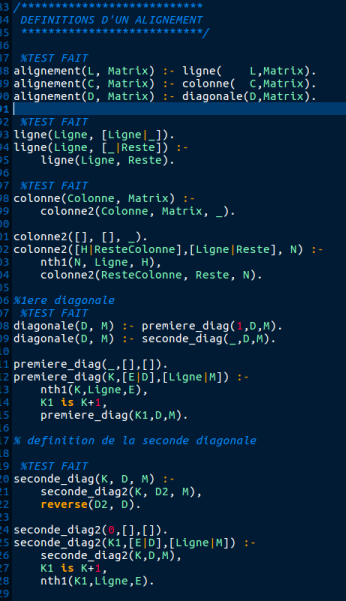
Pour accéder à une ligne, on avance récursivement dans la matrice en récupérant chacune de ses “sous-listes” (cf code ci-dessous)

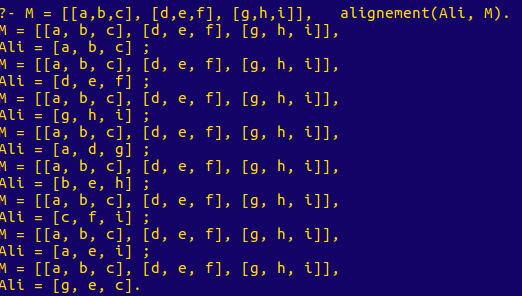
Pour une colonne, nous devons avancer récursivement dans la matrice, et récupérer sur chaque ligne, chaque élément N et l’ajouter à une nouvelle liste où les Nème éléments des autres lignes de la matrice seront aussi ajoutés. (cf code ci-dessous)

Pour récupérer les diagonales, il faut récupérer la première et la deuxième distinctement. Pour la première, nous devons parcourir la matrice sur ce même principe et incrémenter un compteur à chaque étape afin d’ajouter le 1er élément de la première ligne, puis le 2ème élément de la seconde et ainsi de suite jusqu’à N à la même liste qui composera la diagonale.

Pour la seconde, le principe est similaire, mais avant d’incrémenter le compteur il faut d’abord se placer sur la dernière ligne de la matrice et ensuite remonter, afin d’ajouter à une nouvelle liste qui sera la seconde diagonale, le premier élément de la Nème ligne, le second de la N-1ème et le Nème de la première ligne.

On obtiendra ainsi par ces 3 prédicats l’ensemble des alignements possibles. En voici le code :



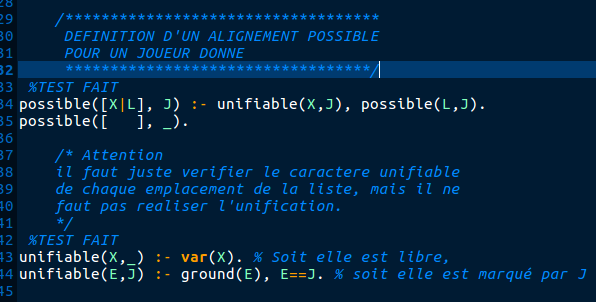


Nous obtenons bien les 8 alignements d’une matrice NxN. Les lignes, puis les colonnes et enfin les diagonales sont obtenues.

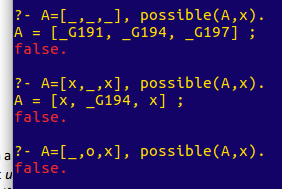
On veut maintenant savoir si pour un alignement A et un joueur J donné, ce joueur J peut encore gagner sur cet alignement, c’est à dire, si toute les cases de l’alignement sont encore libres ou appartiennent déjà à J.

Le prédicat possible va donc avancer récursivement sur chaque case de l’alignement et appeler le prédicat unifiable pour chacune de celle-ci qui renverra true si la variable est libre (prédicat var) ou qu’elle vaut déjà J.

En voici le code et son comportement :



Le comportement suivant est obtenu :

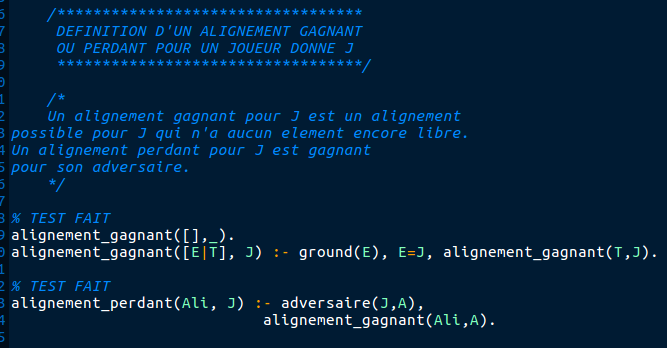


Il est valide car notre première requête nous renvoie une solution pour un alignement avec des variables libres, la 2ème pour un alignement composé de variables libres et du symbole de J et la 3ème, aucune solution car cet alignement n’est plus gagnable pour J. (présence d’un symbole adverse)

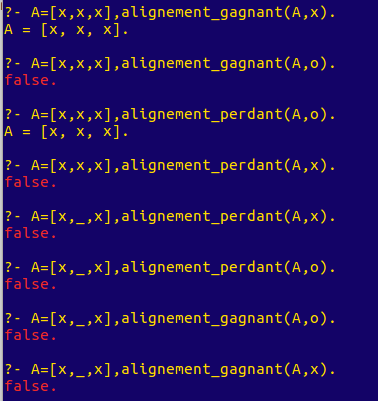
Nous définissons maintenant les prédicats alignement\_gagnant(A,J), alignement\_perdant(A,J) qui valide qu’un alignement est totalement lié à un joueur ou pour perdant, à son adversaire.

Pour l’alignement gagnant : On utilise le prédicat ground qui renvoie true si la variable donnée est totalement instanciée. Et on vérifie qu’elle est instanciée pour J.

Pour l’alignement perdant : Ce sera vrai si l’alignement est gagnant pour l’adversaire.



Utilisation :

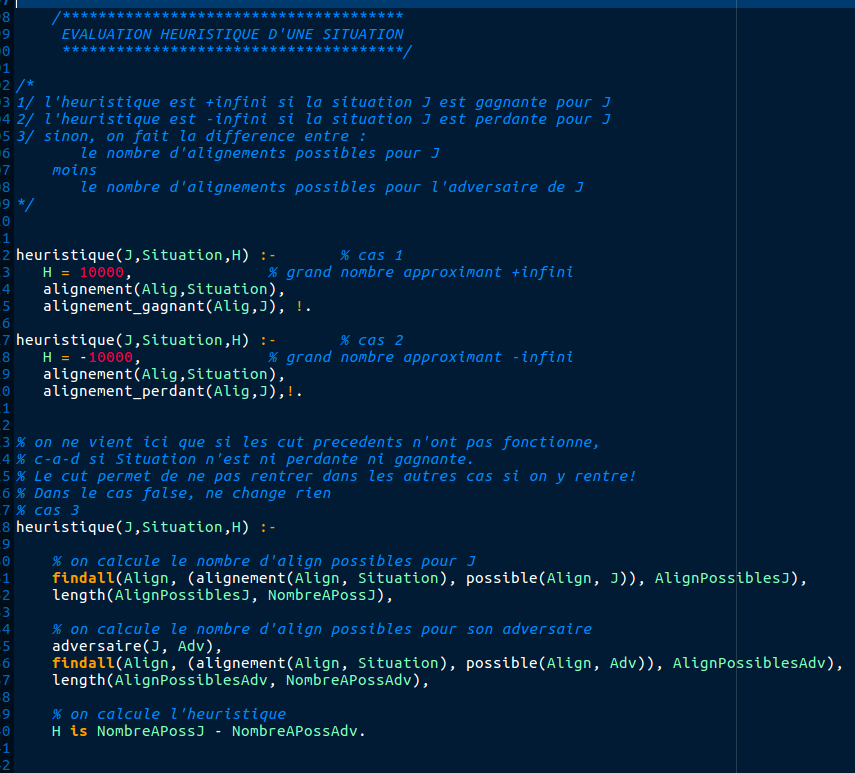


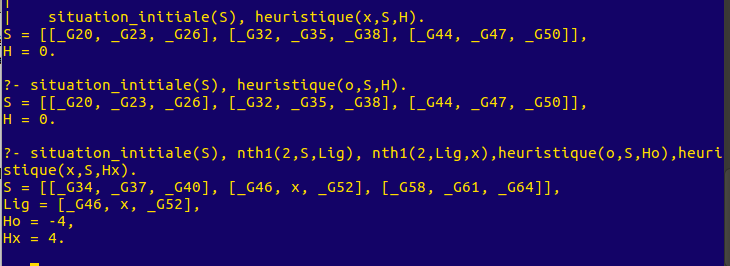
On se rend compte que le prédicat alignement gagnant ne renvoie bien true seulement si l’ensemble des cases de l’alignement sont liées au J donné, et l’alignement perdant, seulement si les cases de l’alignement sont toutes liées à l’adversaire du J donné.

1. **Développement de l’heuristique h**

Nous développons le prédicat heuristique(+Joueur, +Situation, ?H) qui permet de renvoyer la différence entre le nombre d’alignement potentiellement réalisable pour le Joueur dans la situation Situation et ceux réalisables par son adversaire.

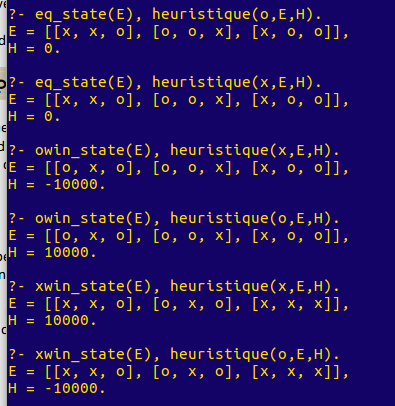
En voici le code commenté :





Sur la capture précédent, on présente les tests réalisés sur l’heuristique. On s’aperçoit donc que l’heuristique est nulle pour les deux joueurs sur la situation initiale, ce qui est normal car tous les alignements restent possibles pour les deux joueurs si aucun symbole n’a été joué.

On fait jouer le joueur x en 2x2 (au milieu) et nous re-testons les heuristiques. L’heuristique de x vaut 4, celle de o, -4. En effet, en jouant au milieu, 4 alignements sont désormais uniquement gagnables par x. (ligne 2, colonne 2 et les deux diagonales). La différence est bien de 4, à l’avantage de x.



On a ici testé que l’heuristique renvoi bien 1000 (infini) pour une situation gagnante pour J, -1000 si c’est son adversaire et 0 en cas d’égalité.

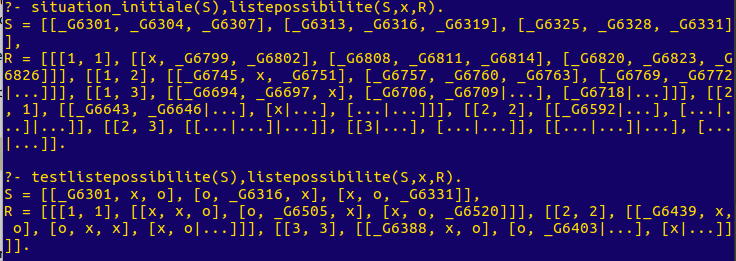
1. **Développement de l’algorithme Negamax**

Nous avons pris connaissance du fichier contenant les explications sur l’implémentation de Negamax. Les commentaires du début de fichier sont clairs. Ils expliquent le but du fonctionnement du prédicat Négamax et ses paramètres, la description de la récursion à appliquer avec ses cas triviaux, et enfin quelles étapes suivre avant de lancer la récursion.

Nous suivons donc cette description comme fil rouge de notre implémentation et implémentons chaque fonction demandée selon les commentaires qui la précède. (Voir directement notre code)



Ceci est le prédicat qui permet de récupérer une liste d’éléments [coordonnées, situation résultante] où chaque élément est un coup possible pour le joueur J dans la situation donnée S.

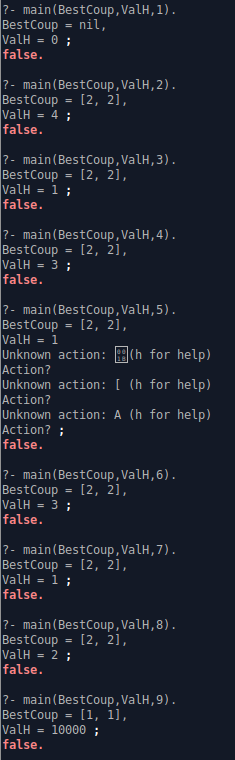


On l’a testé sur la situation initiale qui nous renvoi la liste de toutes les coordonnées et la situation ou un “x” y serait si le joueur x jouait depuis la situation S.

On l’a aussi testé sur une matrice ou seule sont libres les coordonnées (1,1),(2,2) et (3,3).

1. **Expérimentation et extensions**

Voici nos résultats pour des profondeurs maximales de 1 à 9.



Le meilleur coup proposé pour la situation initiale pour une profondeur maximale de 2 à 8 est de jouer au milieu.

Pour une profondeur maximale égale à 1, personne n’a encore joué et l’heuristique est bien de 0. (8 alignements encore possible pour chacun)

Avec une Pmax de 2, on joue au milieu et l’heuristique vaut bien 4 car on a 4 alignements possibles en plus pour celui qui vient de jouer au milieu.

On enchaîne ensuite avec des valeurs heuristiques alternant entre 1 pour une Pmax impaire et 3 pour une Pmax paire. Ceci est dû au fait que sur une Pmax impaire, nous avons alors envisagé que ce soit l’adversaire qui aie joué le dernier et qui aie eu donc l’occasion de “contrer” au mieux le placement du joueur courant et ainsi réduire son nombre d’alignements d’avance. Si la Pmax est paire, le joueur courant est celui envisagé comme le dernier à jouer, et peut donc augmenter son avantage sur le nombre d’alignements. (3 de plus en ce cas, avantage pris en jouant au milieu en début de partie)

Pour la profondeur maximale égale à 9 (la grille complète), la valeur de l’heuristique est 10000 (soit +infini).   
Nous le comprenons comme le fait qu’en jouant ainsi, le joueur J gagnera coûte que coûte, car l’algorithme renvoie +infini pour une situation gagnante pour le joueur courant. Cela est due au fait que via la profondeur 9, toutes les possibilités sont évaluées (Nous avions d’ailleurs un message d’erreur out of local stack sur un autre ordinateur).

Cependant, le résultat renvoyé pour le BestCoup m’intrigue. Commencer à jouer en 1,1 “n’enlève” que 3 possibilités d’alignements à l’adversaire contre 4 en jouant au milieu. Ce placement serait plus efficace pour contrer un premier coup qui a été placé au milieu.

Il est donc peut-être possible qu’une stratégie puisse être de commencer par cet angle mais cela paraît surprenant, l’autre option serait la remontée d’une valeur erronée si l’on veut calculer l’heuristique sur l’ensemble du jeu et non une partie.

4.2 - Pour ne pas développer inutilement des situations symétriques, il ne faudrait pas définir les coups comme étant des coordonnées exactes, mais comme un placement soit au centre, soit dans un angle, soit sur un côté ( non angle). Stocker ainsi les états possiblement obtenus et ainsi éviter le calcul de situations similaires.

4-3 Pour passer au jeu du puissance 4, il faudrait reprendre pas mal de choses. Déjà, le puissance 4 se joue sur une grille 6\*7, ce n’est donc pas une matrice N\*N et les calculs de diagonales sont donc différents.

Aussi, la différence majeure est qu’au puissance 4 on ne peut jouer que sur la ligne la plus basse ou bien sur les jetons ayant déjà été joués. Il faudrait donc revoir la manière donc un joueur peut jouer et comment trouver les alignements possibles sur cette grille.

4-4 Si l’on veut améliorer l’algorithme en élaguant certains coups inutiles via l’algorithme alpha-beta, il faut donc implémenter celui-ci. Pour cela, il est envisageable que chaque état développé possède son alpha et son beta. Il faut ensuite implémenter les règles de transmission père-fils et fils-père vu en cours pour ces 2 valeurs et les appliquées lors de l’exploration de l’arbre. On ajoutera ensuite la clause qui en cas de alpha supérieur à beta, stoppe la recherche sur les autres successeurs de l’état concerné.

**Conclusion :**

Nous avons donc, au cours de ce TP implémenté une version générique de NegaMax pour résoudre tout type de problème modélisable dans un arbre de jeu à 2 joueur pour une Pmax donnée, pour trouver le meilleur coup initial par exemple. Il faut évidemment ajouter à ça un second fichier qui décrit les caractéristiques du jeu modéliser. Ici, c’est le fichier TicTacToe qui permet de décrire la structure des états, des alignements, des possibilités… et ainsi pouvant être utilisé directement par NegaMax. Notre implémentation reste optimisable, certains calculs sont redondants et peuvent être évités comme expliqué précédemment.

Le format générique de NegaMax, nous permettrait aussi de l’implémenter pour d’autres jeux, en créant un fichier fournissant les caractéristiques similaires à celle pour le tictactoe, mais en adéquation avec le jeu souhaité.