Intelligence artificielle pour le jeu de stratégie « Pogo »

[Sous-titre du document]

Luc CADORET, Belkacem LAHOUEL, Tristan GIERCZYNSKI

RESUME

Rapport de projet de l'UV IA41, UTBM, 2014. Implémentation d'une intelligence artificielle pour le Pogo, en Prolog.

utbm

Table des matières

Introduction

Ce projet s'inscrit dans le cadre de l'UV IA41, dans laquelle il nous a été demandé de réaliser un projet incluant une intelligence artificielle. Parmi la liste des sujets proposés, notre choix s'est porté sur un dénommé jeu « Pogo ».

Pogo est un jeu de société assez basique à expliquer, mais comportant quelques subtilités stratégiques, ce qui le rend plutôt intéressant à étudier dans un projet d'intelligence artificielle.

Nous avons donc réalisé ce projet à l'aide de plusieurs outils : Prolog, pour le côté intelligence artificielle, et Qt/C++, pour le côté interface graphique. Malheureusement nous n'avons pas réussi à lier les deux, ce qui nous a contraints à n'utiliser que Prolog. Le programme final comporte deux modes de jeux : IA contre IA, ou Joueur contre IA. Le joueur peut parfaitement choisir sa couleur et le niveau de son adversaire.

Dans ce rapport, nous allons présenter ce projet dans son ensemble. En premier lieu, nous détaillerons les règles du jeu, puis nous reviendrons sur la problématique qu'amène ces mécaniques de jeu. Ensuite, nous détaillerons les outils et algorithmes utilisés pour construire une IA, tels que les méthodes d'évaluation, ou encore les structures de données. Nous terminerons par exposer quelques cas concrets, ainsi que leur résolution.

I – Comment jouer au PoGo

1) Le but

L'état initial d'une partie de Pogo se présente de la façon suivante : neuf cases, 6 pions noirs, 6 pions blancs.

Pour comprendre le but du jeu, il faut comprendre la notion de « couleur de pile ». Au Pogo, une pile appartient à un joueur si le pion qui est tout en haut de cette dernière appartient au joueur. Ainsi, une pile peut comporter des pions noirs ou blancs, mais seul le pion au sommet définira l'appartenance de cette pile.

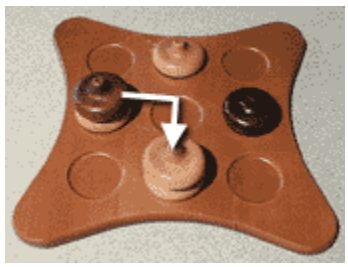
Un joueur ne peut jouer qu'avec les piles qui lui appartiennent. Le but du jeu est donc de posséder toutes les piles du plateau (ou la pile, puisqu'il peut n'en rester qu'une seule).

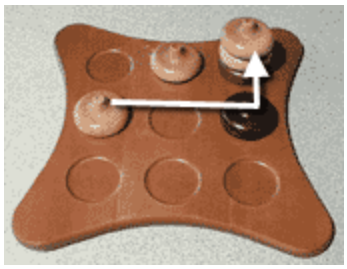
2) Les déplacements autorisés

Pour parvenir à ses fins, les joueurs ont le droit de déplacer leurs pions selon des règles précises :

- On ne peut prendre qu'un, deux ou trois pions en même temps
- On peut déplacer des piles de pions, la longueur du déplacement doit être égale au nombre de pions dans la pile qu'on déplace.
- Lors de ce déplacement, la ligne droite n'est pas forcée : on peut effectuer 1 coude (si 2 ou 3 pièces prises), ou 2 coudes (si 3 pièces prises). Faire deux coudes revient à se déplacer d'une seule case.

Voici les images correspondant aux déplacements possibles :

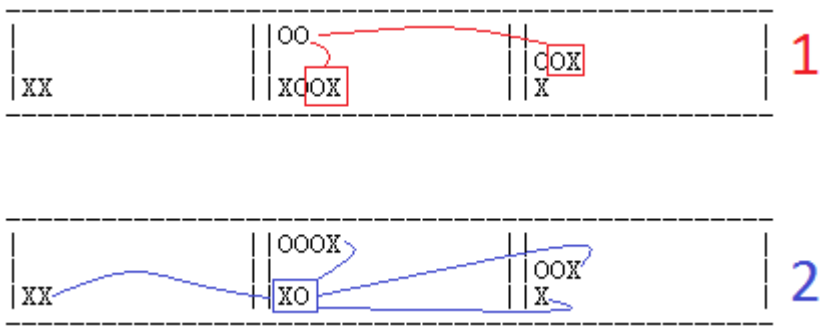




II – Fonctionnement de l'intelligence artificielle

1) Problématique amenée par le jeu

Bien que relativement simples, les règles apportent cependant une très grande dimension stratégique. Evaluer un état du jeu peut s'avérer plus complexe que prévu : il n'y a pas seulement la couleur de la pile qui importe, mais aussi les pièces qui se libèrent quand on déplace la pile. Bien vite, on atteint une problématique : comment évaluer la valeur d'une pile ? Une pile qui m'appartient, mais qui contient beaucoup de pions adverses a-t-elle beaucoup de valeur ? A-t-elle plus de valeur qu'une plus petite pile, qui contient seulement des pions à moi ?



L'exemple ci-dessus est bien représentatif. Les croix, pour prendre la dernière tour des ronds, n'a que deux mouvements possibles. Seulement, ces deux coups libère tous les deux une pile pour les ronds. Les croix font donc le mouvement qui les amène en 2, et libère la pile des ronds. Cette dernière a alors tout un panel de mouvements, peut-être mieux placés que la pile que les ronds avaient avant.

C'est là toute la stratégie du Pogo : parfois, il faut sacrifier une pile pour en obtenir une meilleure mieux placée, parfois il faut essayer de gagner du temps pour forcer l'adversaire à se diviser… il y a énormément de possibilités différentes.

2) La représentation d'un état dans l'implémentation

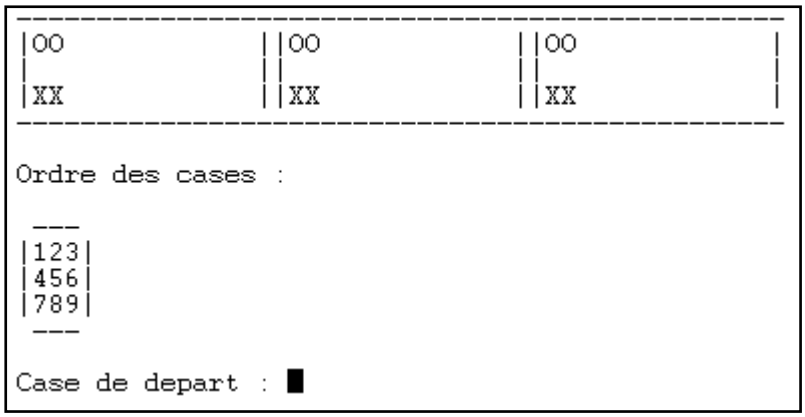
Instinctivement, on pourrait se dire que la façon la plus simple de représenter une pile de pions est une liste, avec le pion le plus en bas en tête de liste, et le pion le plus en haut en queue de liste. C'est d'ailleurs de cette façon que c'est représenté console (premier élément de la liste – si on parle d'ordre chronologique, donc dernier de la pile – est le pion, à gauche, dernier pion à droite).

Dans Prolog, nous avons choisi de représenter cette liste dans le sens inverse (ordre FILO, le premier de la liste est le dernier arrivé), afin de diminuer les calculs : pour chaque case, le premier élément de la liste est le pion tout au-dessus. Cela offre une plus grande rapidité de calcul et nous n'avons pas besoin de parcourir tout la liste à chaque fois :

- l'index du pion qui sera la base de la pile qu'on souhaite déplacer, correspond aussi à la taille de la pile.
- On n'a pas besoin de parcourir toute la pile en partant de la fin pour connaitre quels seront les 1, 2 ou 3 éléments qui seront dans la pile qu'on veut déplacer.

Ce genre d'amélioration, même si elle peut paraître minime, s'avérera bien pratique lorsqu'il faudra effectuer des milliers de calculs d'états.

Voici la représentation de l'état initial dans l'interface console, ainsi que dans le fichier prolog :



L'état initial, vu depuis l'interface console. Pour chaque case, le pion le plus à droite est le pion au sommet de la pile. Le plus à gauche est le fond de la pile.

```
[1,1,-1,1,1,-1,1,1,-1,-1,-1,-1,0,0,-1,0,0,-1,0,0,-1]
```

L'état initial, vu depuis le fichier prolog. Les 1 représentent les pions du joueur blanc, et les 0 ceux du joueur noir. Les -1 servent à indiquer à Prolog que c'est la fin de la liste des pions sur une case, et on passe à la prochaine. L'ordre des cases est défini comme vu dans l'image précédente. Cette structure nous permet de n'avoir à retenir que 21 entiers à chaque fois, ce qui est un réel gain comparé à une structure plus facile à comprendre qui serait un tableau de 3*3*21 éléments (au maximum) ou un tableau redimensionnable éventuellement, qui serait plus embêtant à traiter.

Pour savoir comment passer d'un état à un nouvel état, nous avons mis en place des prédicats qui trouvent ce nouvel état à partir d'un coup. Un coup est un triplet [case de départ, case d'arrivée, indice du pion en partant du dessus de la pile]. C'est d'ailleurs le triplet que doit rentrer le joueur pour jouer.

3) L'évaluation d'un état

L'évaluation d'un état est primordiale pour le bon fonctionnement d'une IA. Nous avons vu plus tôt dans l'explication de la problématique que l'évaluation d'un état était plutôt compliquée. C'est pourquoi nous avons pensé à intégrer plusieurs types d'évaluations différentes, proposant ainsi un niveau ou un style de jeu différent.

Pour la suite des explications, une tour sera dite contrôlée par un joueur J, si le pion au sommet de celle-ci appartient au joueur J. De plus, on dira qu'un pion contrôle efficacement une pile, s'il a beaucoup de pions adverses en dessous de lui. La mobilité d'un joueur est le nombre de coups qu'il peut jouer. Nos fonctions ne renvoient pas un nombre entre -1 et 1, mais cela n'a pas d'importance : il s'agit d'une simple homothétie qui élargit (de manière plus ou moins proportionnelle) l'intervalle d'évaluations.

Nous avons, pour la fonction d'évaluation, retenu plusieurs critères :

- la mobilité
- le nombre de tours contrôlées
- l'efficacité de contrôle d'une pile.

On se rend compte que ces différents critères, quoique très différents, se recoupent. Par exemple :

- La mobilité d'un joueur adverse est affectée de la même façon qu'un de nos pions contrôle efficacement une tour adverse, plus ou moins
- Le nombre de tours contrôlées élevé est synonyme de mobilité, et réciproquement
- …

Nous pourrions peut-être trouver d'autres fonctions d'évaluation, et d'autres critères. Ce sont les critères que nous avons retenus comme étant les plus importants. Encore une fois, l'évaluation d'un état est une chose plutôt difficile, nous nous en sommes rendus compte.

Ces trois critères nous permettront d'établir 7 fonctions d'évaluation d'un état donné.

eval0

Elle calcule une évaluation en fonction du nombre de piles contrôlées par chacun des joueurs. C'est une des fonctions les plus simples mais également une des plus efficaces que nous avons crées.

eval1

Dans cette fonction d'évaluation, nous sommons les évaluations des quatre premiers pions de chaque pile. Nous voulions évaluer l'effet qu'aurait chaque mouvement sur l'état suivant : on peut bouger de 0 à 3 pions, nous évaluons donc les pions en positions 1 à 4. Mais ce fut une mauvaise idée, elle est visiblement une des pires que nous avons faites.

eval2

Nous raisonnons de la même manière que pour eval1, sauf que nous donnons un poids prépondérant au pion du sommet. Cela revient en fait à tendre vers eval0, qui considère juste les évaluations des pions aux sommets (donc qui donne une valeur en fonction du nombre de tours contrôlées par chacun). Grâce à ce poids prépondérant, de toute manière, on aura toujours une évaluation qui est influencée par le premier pion, et au pire, on aura une évaluation de 1 (si seul le premier pion nous appartient, étant donné son poids de 4). Elle est donc intéressante, mais pas très mauvaise.

eval3

Cette évaluation nous donne une combinaison linéaire entre le nombre de tours contrôlées et le nombre de pions que l'on peut bouger actuellement. On se rend compte que suivant les poids affectées dans cette combinaison linéaire, cela revient à choisir soit d'aller vers eval0 (nombre de tours contrôlées) soit vers eval4 (que nous détaillerons juste après). Nous ne la retiendrons pas.

eval4

eval4 est la première fonction d'évaluation vraiment forte que nous avons obtenue. Elle permet de voir des choix intéressants, et elle peut même nous battre de temps à autres ! Elle cherche à compter la mobilité d'un joueur, soit le nombre de coups qu'il peut effectuer. De cette façon, elle tend à aller vers des cas où la mobilité du joueur est maximisée, tout en minimisant la mobilité adverse.

eval5

eval5 nous donne l'efficacité de contrôle d'un pion sur sa tour : plus il aura de pions adverses en dessous de lui, plus il sera efficace. Cette fonction est également très intéressante, et on se rend compte qu'elle permet de gagner très rapidement : en effet, lors de parties IA vs IA, c'est celle qui permet de sortir vainqueur en un minimum de coups.

eval6

Cette fonction d'évaluation calcule une combinaison linéaire sur les trois paramètres essentiels, à savoir : l'efficacité de contrôle d'un pion sur sa tour, le nombre de tours contrôlées, et le nombre de coups possibles.

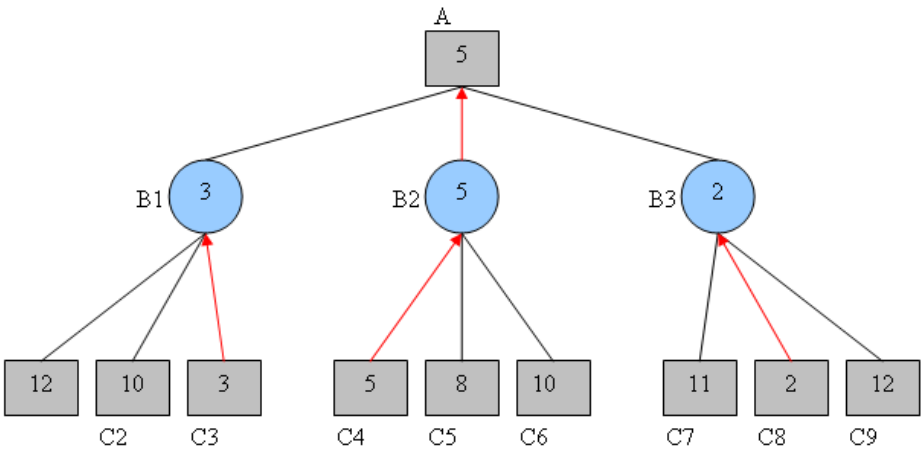
Elle permet également d'avoir de bons résultats. Nous avons essayé d'ajuster les poids, car nous avons le problème de la proportionnalité. Certaines évaluations renvoient des valeurs sur des intervalles plus larges que d'autres…

Pour améliorer les évaluations, nous avons pensé à un genre d'algorithme qui utiliserait des fonctions génétiques : on améliore la fonction d'évaluation, en ajustant les poids après un certain nombre d'appels (exemple : +1 ou -1 en fonction du résultat, donc succès ou échec, que nous aura permis d'obtenir cette fonction d'évaluation). Certes, ce n'est pas un modèle rigoureux, mais il nous aurait permis d'approcher de manière empirique la fonction d'évaluation optimale… Nous avons également pensé à réaliser une base de faits, sur certains cas évidents, afin de personnaliser le jeu de l'intelligence artificielle, lui donner un peu d' « humanisme ». Pour finir, nous avons pensé à ajouter une évaluation « spéciale » pour l'état de succès : on attribue une valeur très grande à nos état de succès, qu'elles ne peuvent calculer en état « normal ». On pondère également par la profondeur actuelle, mais nous verrons cela plus tard, dans la partie sur le Minmax.

4) L'algorithme Minmax

a) Son fonctionnement

Pour parvenir à faire une IA correcte, nous avons décidé d'implémenter l'algorithme Minmax, relativement simple à comprendre. Il consiste à descendre tout en bas de l'arborescence (jusqu'au moment où il n'est plus possible d'obtenir un nouvel état, OU si on a atteint la profondeur renseignée), puis faire remonter le meilleur état jusqu'à la racine, sachant qu'un des joueurs va décider de minimiser l'évaluation de cet état (dans notre programme, les noirs), et l'autre va chercher à la maximiser (les blancs). En voici un exemple (les carrés maximisent, les ronds minimisent) :



Les B sont choisis par rapport aux C minimums, et le A est choisi par rapport au B maximum.

Cet algorithme, bien que simple à comprendre, nous a été plutôt difficile à implémenter, à cause du fonctionnement de prolog : uniquement récursif, impossible de faire une simple boucle « for » pour trouver les nœuds à l'évaluation minimum ou maximum. Mais nous avons finalement réussi à l'adapter à notre programme. Il nous a été également compliqué de le débuguer, puisque trouver la branche précise qui empêche le bon fonctionnement du prédicat relève de beaucoup de patience : un joueur peut avoir plus ou

moins de 16 coups possibles (à l'état initial). On atteint donc rapidement un nombre de branches énormes, et encore plus de calculs.

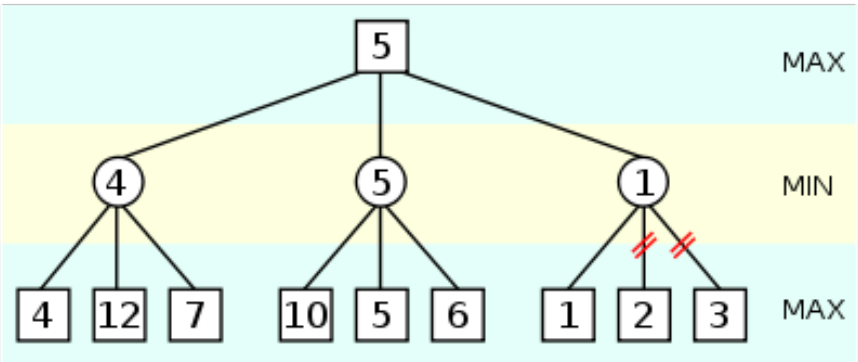
Nous avons cherché à implémenter la simplification de Minmax, Négamax. Quelques test se sont révélés très concluants (plus rapide que Minmax), cependant nous ne sommes pas sûrs que Négamax soit applicable à notre jeu, puisqu'il réclame certaines conditions. Par précaution, nous avons donc gardé Minmax.

A noter que dans notre Minmax, lorsque l'évaluation de deux états ont la même valeur, le programme ne vas pas sans arrêt choisir la même : il va en choisir une des deux au hasard. Cette décision a été prise pour limiter les boucles infinies (les deux joueurs se stabilisent parfaitement entre eux), et les parties tout le temps similaires.

De plus, nous avons mis en place une pénalisation de l'état de victoire par la profondeur (plus la victoire est proche, et donc moins profonde, meilleur sera l'état de victoire). Grâce à ce système, l'intelligence artificielle, si elle trouve deux états de victoire, elle ira à celui qui est en moins de coups.

b) Rapidité d'exécution, grâce à l'élagage alpha-bêta

L'algorithme Minmax n'est rien sans son élagage alpha-bêta, qui permet d'augmenter les performances de ce dernier, en y greffant un questionnement simple « Si je sais que cette branche est déjà pire qu'une branche que je connais déjà, pourquoi continuer à la visiter ? ». Couper une branche nous évitera donc de faire des calculs supplémentaires, sans changer le résultat de Minmax. Voici un exemple ci-dessous :



Là encore, il n'a pas été facile d'implémenter cet élagage, mais nous avons finalement réussi à l'ajouter à Minmax.

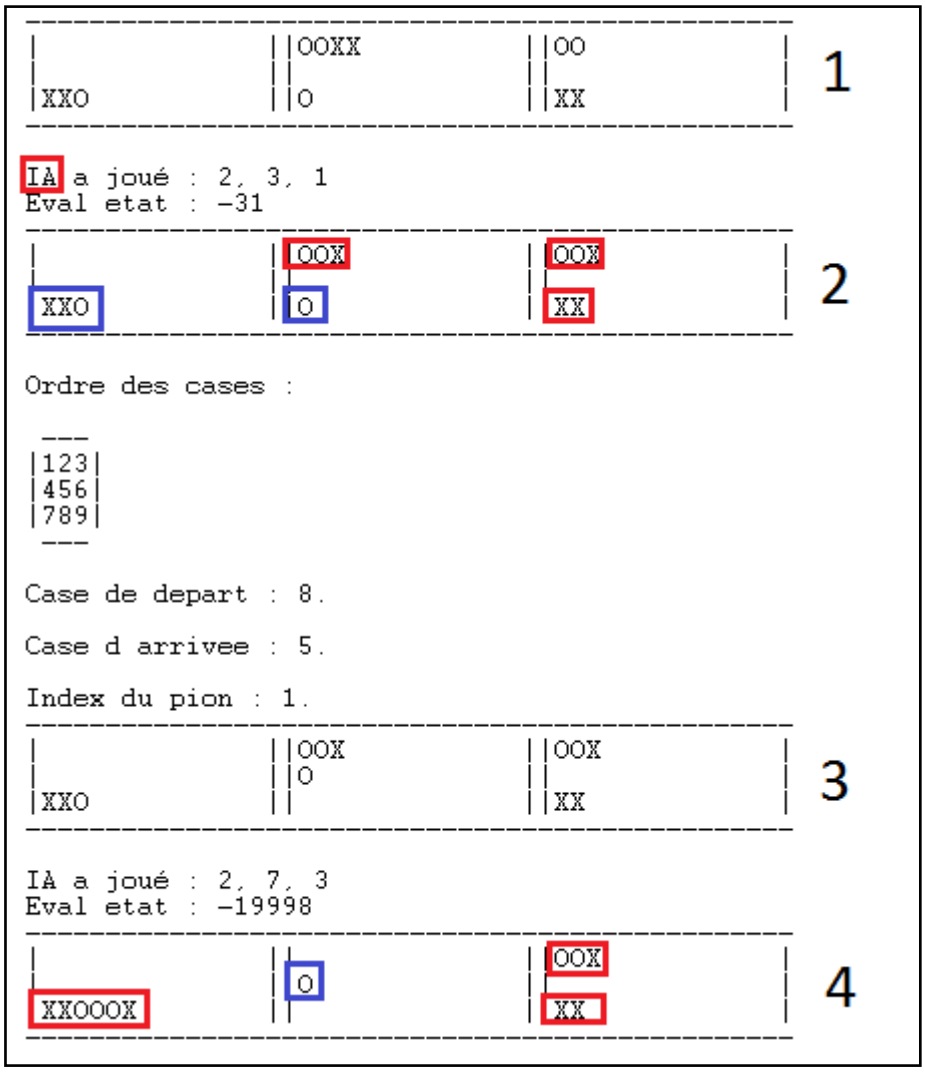
Là où Minmax sans élagage possède une complexité de $\mathbf{b}^{\mathbf{d}}$, avec **b** facteur de ramification, et **d** la profondeur, un Minmax avec élagage possède une complexité (en moyenne) de $\mathbf{b}^{\mathbf{3d/4}}$, ce qui n'est pas négligeable pour les grosses profondeurs.

III – Quelques situations concrètes

Remarque : on se réfèrera souvent aux fonctions d'évaluations comme étant des intelligences artificielles à part entière. C'est en effet ces fonctions d'évaluation qui changent complètement le comportement de l'IA…

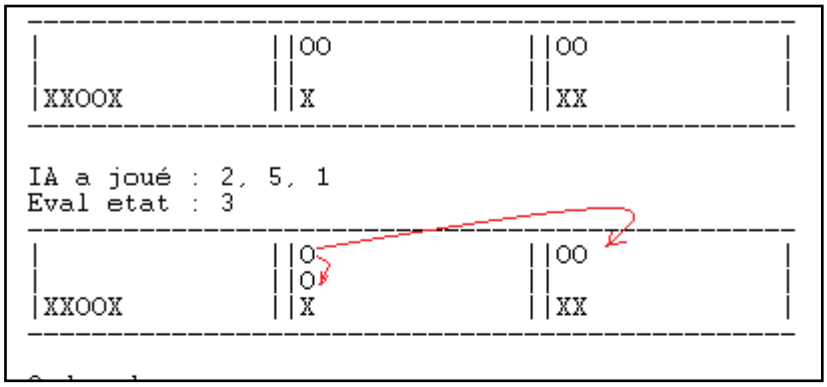
Voici quelques situations concrètes pour chacune des évaluations, montrant la façon dont fonctionnent les différentes fonctions d'évaluations :

eval4 :

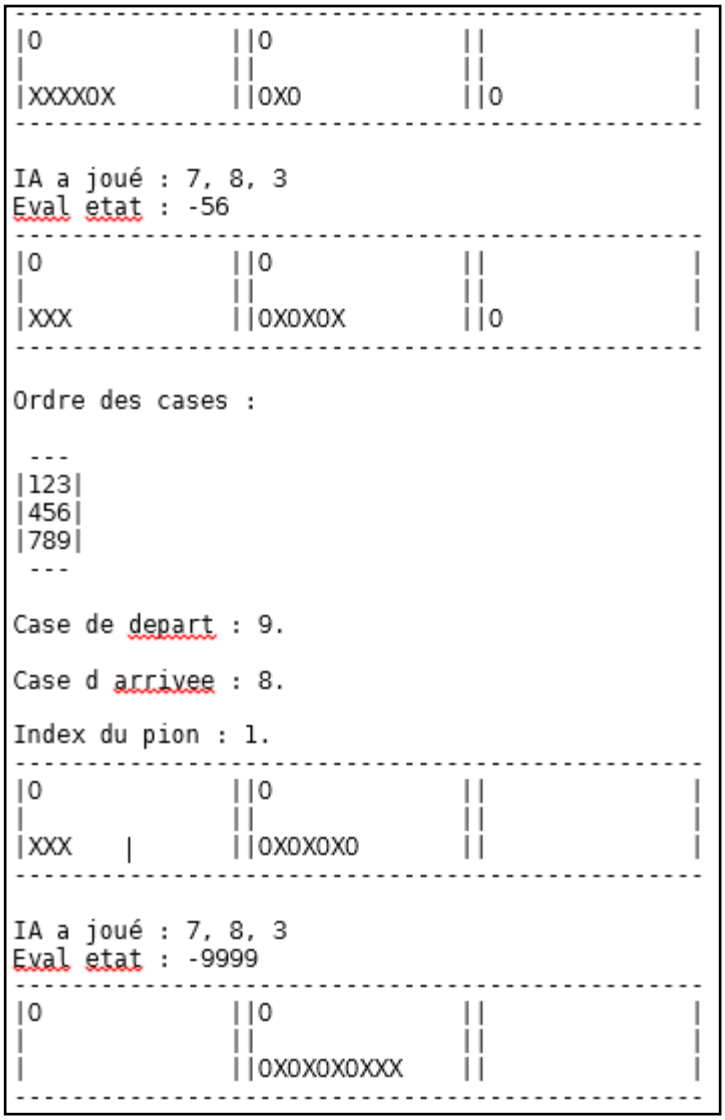


Dans ce genre de situations, on se rend bien compte que l'IA eval6 (les croix, en rouge) essaye de minimiser les déplacements du joueur (les ronds). On se rend compte aussi qu'en supprimant petit à petit l'amplitude des coups du joueur adverse, il devient totalement à la merci de l'IA. Au stade 4, l'IA a gagné, quoi que fasse le joueur.

eval0 :



eval0 va souvent chercher à maximiser son nombre de piles pour augmenter son évaluation. On voit (2ème image, les blancs) que c'est une bonne méthode pour avancer petit à petit ses pions tout en se faisant protéger par ses voisins. De cette manière, elle peut parfois faire des choix ridicules, c'est pourquoi elle est classée comme étant le niveau « facile » du jeu… Mais elle reste tout de même intéressante et c'est un critère primordial, non négligeable.



eval6 :

eval6 va avoir un comportement assez variable, du fait de la combinaison linéaire entre les trois évaluations qui la composent. Dans cet exemple, nous serons les ronds (O) et eval6 aura le rôle de l'IA avec les croix (X). Elle permet de tenir compte de tous ces paramètres en même temps, ce qui en fait une IA assez imprévisible (on ne sait pas, parfois, si elle va préférer un coup et pour quelles raisons… Cela dépendra sûrement des coefficients affectés). Le coefficient de 10 devant eval0 lui permettra de séparer ses tours à des moments de jeu stable, pour pouvoir tendre des pièges au joueur adverse. Elle est également redoutable, elle peut contrôler une tour très concentrée en pions adverses, assez efficacement…

Dans l'exemple qui suit, on peut voir que eval6 peut nous forcer à jouer certains coups, afin de pouvoir gagner par la suite. Dans la position finale, la victoire est quasi-certaine, nous n'avons que peu de possibilités de mouvements, et encore moins de mouvements qui nous laissent dans la course…

IV – Améliorations possibles

Evidemment, l'application finale est loin d'être parfaite, et nous avons décelé plusieurs améliorations possibles :

- Même si un peu d'aléatoire a été intégré, il arrive toujours que deux IA trouve un état d'équilibre entre elles (en particulier lorsqu'elles ont la même fonction d'évaluation), et qu'elles s'annulent l'un l'autre coup après coup : on obtient une boucle infinie sans jamais voir la fin du jeu. Pour pallier à cela, on pourrait implémenter un compteur évitant aux IA de refaire X fois la même boucle, ou encore retenir les coups faits précédemment et l'obliger à ne pas en refaire certains.
- Une meilleure fonction d'évaluation est aussi à trouver, puisqu'il y a toujours moyen de l'améliorer. On pourrait par exemple ajouter des bases de faits pour l'ouverture du jeu, mais cela demanderait de beaucoup s'attarder sur le jeu.

Conclusion

Ce projet a été intéressant à réaliser sur plusieurs aspects :

- travail en groupe,
- coordination,
- utilisation de nouveaux outils de travail,
- …

Tout d'abord, il nous a permis de réaliser qu'un jeu aux règles élémentaires peut cacher une certaine complexité, et que l'identification et la résolution des problèmes qui vont avec n'est pas toujours facile. Il nous a donc fallu tenter plusieurs approches pour aboutir aux résultats finaux.

Nous avons donc choisi d'implémenter Minmax, et son élagage alpha-beta, ce qui nous a permis de voir sur des situations concrètes comment l'algorithme fonctionnait précisément. Bien que difficile à implémenter en Prolog, nous sommes finalement satisfaits de son efficacité et de sa rapidité.

Pendant la phase de développement, nous avons perdu beaucoup de temps à réaliser une interface graphique visuelle agréable, avec Qt/C++. Malheureusement, alors que cette dernière était fonctionnelle, il nous a été impossible de la lier avec notre fichier Prolog. Plusieurs dizaines d'heures ont été perdues, incluant le débogage et la re-conception d'une interface minimaliste en Prolog. Cela reste la principale déception de notre projet.

Mis à part cela, nous avons correctement su nous organiser, et cela a été une occasion de plus pour apprendre à mieux travailler en groupe. Nous considérons le pari comme réussi, puisque l'application finale permet de jouer contre IA de façon fonctionnelle, et peut même nous batte !

Annexes

1) Prédicat Minmax avec élagage alpha-bêta :

Voici les prédicats nécessaires au bon fonctionnement du Minmax avec l'élagage alpha-bêta.

```
% minmax(+ETAT,+JOUEUR,+DEPTH,-COUP, -EVALETAT +LEVEL)
% minmax prend l'état actuel, ainsi que le joueur qui doit jouer, et ressort le
meilleur coup que doit joueur JOUEUR
% la profondeur de la recherche est caractérisée par DEPTH. On donne des
profondeurs différentes en fonction du niveau du joueur.
% EVALETAT est là en particulier pour le debug

minmax(ETAT,JOUEUR,BESTCOUP,EVALETAT,LEVEL):-
                        (LEVEL = 0,!, DEPTH = 3, alphabeta(ETAT,JOUEUR,LEVEL,-
100000,100000,BESTCOUP,EVALETAT,DEPTH);
                        LEVEL = 1,!, DEPTH = 3, alphabeta(ETAT,JOUEUR,LEVEL,-
100000,100000,BESTCOUP,EVALETAT,DEPTH);
                        LEVEL = 2, DEPTH = 4, alphabeta(ETAT,JOUEUR,LEVEL,-
100000,100000,BESTCOUP,EVALETAT,DEPTH)).
                         % -100000 et 100000 sont des valeurs excessivement
grandes pour simuler +inf et -inf
```

```
% alphabeta(+ETAT,+JOUEUR,+ALPHA,+BETA,?BESTCOUP,?BESTEVAL,+DEPTH)
% effectue un minmax avec élagage alpha-beta
% la décrémentation de la profondeur, recherche des coups possibles pour le
joueur, et on
% continue en profondeur tant qu'on peut

alphabeta(ETAT, JOUEUR, LEVEL, Alpha, Beta, BESTCOUP, BESTEVAL, Depth) :-
      Depth > 0,
      OneDeeper is Depth - 1,
      coups_possibles_joueur(ETAT, JOUEUR, L),
      length(L,LMOVES),
      LMOVES > 0,
      !,
      boundedbest(ETAT,L,LEVEL, Alpha, Beta, JOUEUR, OneDeeper, BESTCOUP,
BESTEVAL).

alphabeta(ETAT, _,LEVEL, _, _, _, Val, 0) :- % Profondeur atteinte, on évalue la
feuille
            eval(ETAT,Val,LEVEL),
            (Val = 9999, Val1 is Val*1,!;
            Val = -9999, Val1 is Val*1,!;
            Val = Val1).

alphabeta(ETAT, _,LEVEL, _, _, _, Val, _) :- % Si plus de coup avant d'avoir
atteint la profondeur 0
            eval(ETAT,Val,LEVEL),
            (Val = 9999, Val1 is Val*(DEPTH+1),!; % Pénalisation par la
profondeur
            Val = -9999, Val1 is Val*(DEPTH+1),!;
            Val = Val1).
```

```
% boundedbest(+ETAT,+COUPLISTE,+LEVEL, +Alpha, +Beta, +JOUEUR, +Depth, ?BESTCOUP,
?BESTVAL)
% la fonction boundedbest va s'occuper de faire le lien entre alphabeta (qui
évalue les branches)
% et goodenough (qui est chargée de comparer ces branches)

boundedbest(ETAT,[[D,A,I]|NEXTCOUPS],LEVEL, Alpha, Beta, JOUEUR, Depth, BESTCOUP,
BESTVAL) :-
  inverser_joueur(JOUEUR, J2),
  nouvel_etat(ETAT,D,A,I,NEXTETAT),
  alphabeta(NEXTETAT, J2,LEVEL, Alpha, Beta, _BESTCOUP, Val, Depth),
  goodenough(ETAT,NEXTCOUPS,LEVEL,Depth, Alpha, Beta, JOUEUR, [D,A,I], Val,
BESTCOUP, BESTVAL).
```

```
% goodenough(+ETAT,+MOVELIST,+LEVEL,+Depth, +Alpha, +Beta, +JOUEUR, +COUP, Val,
BESTCOUP, BESTEVAL)
% goodenough est le prédicat chargé d'évaluer de comparer les évaluations qu'il
reçoit avec
% alpha et beta. C'est lui qui se charge de cut (de l'élagage)

goodenough(_,[],_,_, _, _, _, COUP, Val, COUP, Val) :-!.   % On a fini la liste
de coups

goodenough(_,_,_,_, Alpha, Beta, JOUEUR, COUP, Val, COUP, Val) :- % Cas de cut
  JOUEUR = 0, Val > Beta, !                  % MIN a dépassé beta
  ;
  JOUEUR = 1, Val < Alpha, !.                % MAX est passé sous alpha

goodenough(ETAT,MOVELIST,LEVEL,Depth, Alpha, Beta, JOUEUR, COUP, Val, BESTCOUP,
BESTEVAL)  :-
  newbounds(Alpha, Beta, JOUEUR, Val, NewAlpha, NewBeta),
  boundedbest(ETAT,MOVELIST,LEVEL, NewAlpha, NewBeta, JOUEUR, Depth, COUP1,
Val1),
  betterof(JOUEUR, COUP, Val, COUP1, Val1, BESTCOUP, BESTEVAL).
```

```
% newbounds(+Alpha, +Beta, +JOUEUR, +Val, -NewAlpha, -NewBeta)
% ce prédicat est chargé de mettre à jour Alpha ou Beta selon les conditions
rencontrées

newbounds(Alpha, Beta, JOUEUR, Val, Val, Beta)  :-
  JOUEUR = 0, Val > Alpha, !.                % Pour MIN, Lower bond augmente

newbounds(Alpha, Beta, JOUEUR, Val, Alpha, Val)  :-
  JOUEUR = 1, Val < Beta, !.                 % Pour MAX, upper bond diminue

newbounds(Alpha, Beta, _, _, Alpha, Beta).           % Rien ne change
```

```
% betterof(+JOUEUR, +COUP1, +Val1, +COUP2, +Val2, -BESTCOUP, -BESTEVAL)
% Choisit le meilleur coup entre COUP1 et COUP2, et le stocke dans BESTCOUP (et
BESTEVAL)
% Si les coups sont égaux, on choisit l'un des deux au hasard.

betterof(JOUEUR, COUP1, Val1, _, Val2, COUP1, Val1)  :-  % COUP1 est meilleur que
COUP2
  JOUEUR = 1, Val1 > Val2, !       % rappel : pour MAX (1), on cherche la valeur
la plus HAUTE !!
```

```
  ;
  JOUEUR = 0, Val1 < Val2, !.

betterof(_, COUP1, Val1, COUP2, Val2, RANDCOUP, RANDVAL)  :-  % COUP1 et COUP2
sont égaux : on fait en random
  Val1 = Val2,
  random(0,2,R),
  (R = 0, RANDCOUP = COUP1, RANDVAL = Val1,!;
   R = 1, RANDCOUP = COUP2, RANDVAL = Val2).

betterof(_, _, _, COUP2, Val2, COUP2, Val2).     % sinon COUP2 est meilleur

% inverser_joueur(+J1,-J2).
% transforme J1 = 1 en J2 = 0 et vice versa

inverser_joueur(1,0).
inverser_joueur(0,1).
```

2) Fonctions d'évaluation :