# Intelligence artificielle pour le jeu de stratégie Pogo

-

Rapport de projet de l’UV IA41



Luc CADORET

Tristan GIERCZYNSKI

Belkacem LAHOUEL

## SOMMAIRE

[Introduction 3](#_Toc390355139)

[I – Comment jouer à PoGo 4](#_Toc390355140)

[1) Le but 4](#_Toc390355141)

[2) Les déplacements autorisés 4](#_Toc390355142)

[II – Fonctionnement de l’intelligence artificielle 6](#_Toc390355143)

[1) Problématique amenée par le jeu 6](#_Toc390355144)

[2) La représentation d’un état au sein du fichier prolog 7](#_Toc390355145)

[3) L’évaluation d’un état 8](#_Toc390355146)

[4) L’algorithme minmax 8](#_Toc390355147)

[a) Son fonctionnement 8](#_Toc390355148)

[b) Rapidité d’exécution, grâce à l’élagage alpha-bêta 9](#_Toc390355149)

[III – Quelques situations concrètes 11](#_Toc390355150)

[IV – Améliorations possibles 12](#_Toc390355151)

[Conclusion 13](#_Toc390355152)

[Annexes 14](#_Toc390355153)

# 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 à PoGo

### 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 ». A 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.

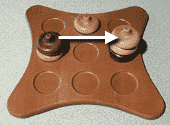
Un joueur ne peut jouer qu’avec les piles qui lui appartiennent. Le but du jeu est donc de posséder toutes les piles du plateau (ou la pile, puisqu’il peut n’en rester qu’une seule).

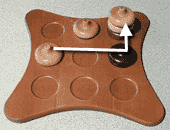
### Les déplacements autorisés

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 :

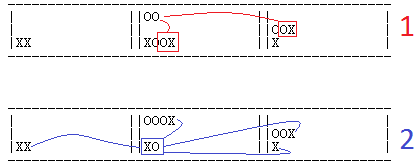
  

# II – Fonctionnement de l’intelligence artificielle

## 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 retirer 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.

## La représentation d’un état au sein du fichier prolog

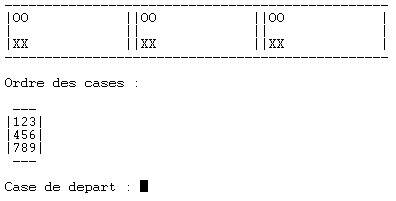
Instinctivement, on pourrait se dire que la façon la plus simple de représenter une pile de pions est une liste, avec le pion le plus en bas en tête de liste, et le pion le plus en haut en queue de liste. C’est d’ailleurs de cette façon que c’est représenté console (premier pion à gauche, dernier pion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

Dans Prolog, nous avons choisi de représenter cette liste dans le sens inverse, 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 :

* 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.

C:\Users\Luc\Pictures\prologinit.png

L’état initial, vu depuis le fichier prolog. Les 1 représentent les pions du joueur blanc, les 0 ceux du joueur noir. Les -1 servent à indiquer à prolog que c’est la fin de la liste des pions sur la case.

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.

## 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 n’état était plutôt compliquée. C’est pourquoi nous avons pensé à intégrer plusieurs types d’évaluations différentes, proposant ainsi un niveau de jeu différent. Voici ci-dessous les trois principales :

Niveau variable : eval4. Eval4 est une fonction d’évaluation qui va chercher à maximiser la différence entre ses coups et ceux de l’adversaire. Si elle voit l’état de victoire, elle y ira sans se poser la question de maximiser ses coups.

Pour cette évaluation, on observe un niveau plutôt variable : tantôt des coups très bien pensés, qui bloquent les déplacements de l’adversaire, tantôt elle tombe dans des pièges évidents. Nous n’arrivons pas à déterminer le niveau de cette évaluation par rapport aux autres.

Niveau moyen : eval0. Cette fonction d’évaluation va chercher à maximiser la différence entre les tours qu’elle possède et celles de l’adversaire. Si elle voit l’état de victoire, elle va directement le chercher.

Niveau moyen : eval0. Cette fonction d’évaluation va chercher à maximiser la différence entre les tours qu’elle possède et celles de l’adversaire. Si elle voit l’état de victoire, elle va directement le chercher.

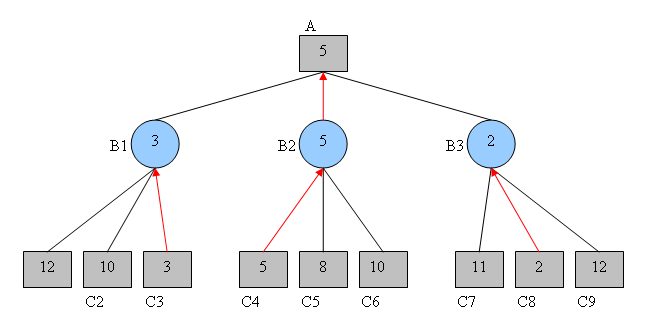
Niveau difficile : eval

TODO BELKA

## L’algorithme minmax

### 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.

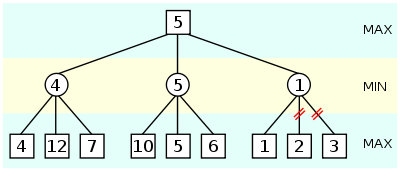
Cet algorithme, bien que simple à comprendre, nous a été plutôt difficile à implémenter, à cause du fonctionnement de prolog : uniquement récursif, impossible de faire une simple boucle « for » pour trouver les nœuds à l’évaluation minimum ou maximum. Mais nous avons finalement réussi à l’adapter à notre 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.

### 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 **bd**, avec **b** facteur de ramification, et **d** la profondeur, un Minmax avec élagage possède une complexité (en moyenne) de **b3d/4**, ce qui n’est pas négligeable pour les grosses profondeurs.

# III – Quelques situations concrètes

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

# 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.

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, grâce à 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.

# Annexes

## Prédicat Minmax avec 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,-10000,10000,BESTCOUP,EVALETAT,DEPTH);

LEVEL = 1,!, DEPTH = 3, alphabeta(ETAT,JOUEUR,LEVEL,-10000,10000,BESTCOUP,EVALETAT,DEPTH);

LEVEL = 2, DEPTH = 4, alphabeta(ETAT,JOUEUR,LEVEL,-10000,10000,BESTCOUP,EVALETAT,DEPTH)).

% -10000 et 10000 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),!.

alphabeta(ETAT, \_,LEVEL, \_, \_, \_, Val, \_) :- % Si plus de coup avant d'avoir atteint la profondeur 0

eval(ETAT,Val,LEVEL).

% 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).

## Fonctions d’évaluation :