INSTITUT NATIONAL DES SCIENCES APPLIQUÉES TOULOUSE
135, Avenue de Rangueil
31077 Toulouse Cedex 4

Compte rendu de TP

Intelligence Artificielle

15 mars 2021

HOK Jean-Rémy
INSA* Promotion 55, 4ème Année IR*

*INSA : Institut National des Sciences Appliquées
*IR : Informatique et Réseaux

INSA Toulouse

135, Avenue de Rangueil

31077 Toulouse Cedex 4

Compte rendu de TP

Intelligence Artificielle

15 mars 2021

HOK Jean-Rémy

INSA* Promotion 55, 4ème Année IR*

*INSA : Institut National des Sciences Appliquées

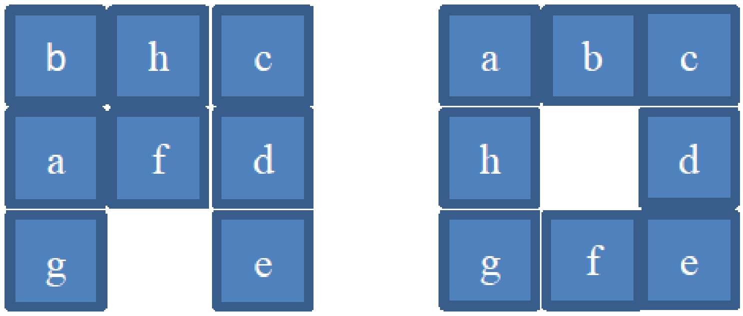
*IR : Informatique et Réseaux

Table des matières

Introduction

L'objectif des deux TPs a été de mettre en pratique nos connaissances sur les algorithmes de résolution de problèmes basés sur la recherche arborescente avec heuristique, à savoir :

- TP1 : Algorithme A* appliqué au Taquin

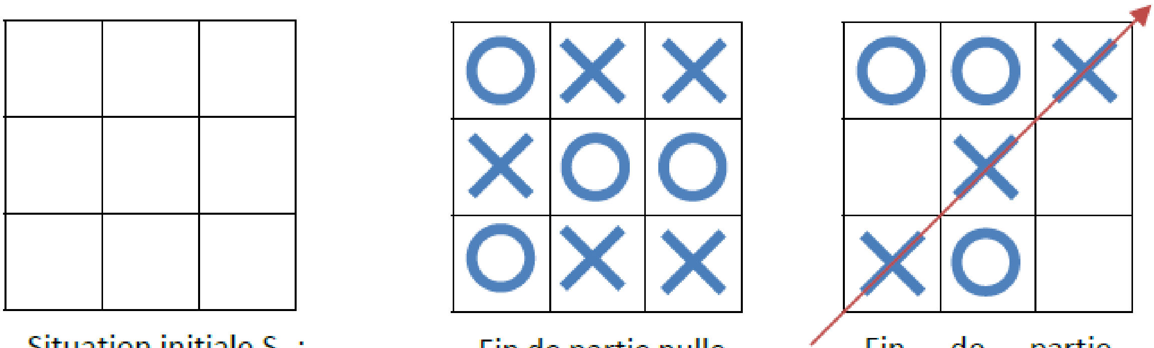


Taquin 3×3
Situation initiale U_0

Taquin 3×3
Situation finale *F*

Figure 1 : Contexte Taquin

- TP2 : Algorithme MinMax / Alpha-Beta appliqué au TicTacToe



Situation initiale S_0 :
le joueur × doit commencer.

Fin de partie nulle

Fin de partie gagnante pour ×

Figure 2 : Contexte TicTacToe

Les fichiers complets des 2 TPs sont récupérables sur un dépôt github : https://github.com/PiKouri/4a-tps-ia.

TP1 – Algorithme A* – Application au Taquin

Familiarisation avec le problème du Taquin 3x3

Quelle clause Prolog permettrait de représenter la situation finale du Taquin 4x4 ?

- `final_state([[1,2,3,4],[5,6,7,8],[9,10,11,12],[13,14,15,vide]]).`

A quelles questions permettent de répondre les requêtes suivantes :

- `initial_state(Ini), nth1(L,Ini,Ligne), nth1(C,Ligne, d).`

On récupère la ligne L et la colonne C de la pièce « d » dans l'état initial : à savoir ligne 2 colonne 3.

- `final_state(Fin), nth1(3,Fin,Ligne), nth1(2,Ligne,P).`

On récupère la pièce P en position (3,2), ligne 3 colonne 2, dans l'état final : à savoir « f »

Quelle requête Prolog permettrait de savoir si une pièce donnée P (ex : a) est bien placée dans U0 (par rapport à F) ?

U0 : état initial et F : état final

```
initial_state(Init), nth1(L,Init,Ligne), nth1(C,Ligne, a),
final_state(Fin), nth1(L,Fin,Ligne2), nth1(C,Ligne2,a).
```

Qui retourne *false* dans notre cas.

Quelle requête permet de trouver une situation suivante de l'état initial du Taquin 3×3 (3 sont possibles) ?

```
initial_state(Init), rule(_, _, Init, Suiv).
```

Quelle requête permet d'avoir ces 3 réponses regroupées dans une liste ?

```
initial_state(Init), findall(Suiv, rule(_,_,Init,Suiv),Suiv_Liste).
```

Quelle requête permet d'avoir la liste de tous les couples [A, S] tels que S est la situation qui résulte de l'action A en U0 ?

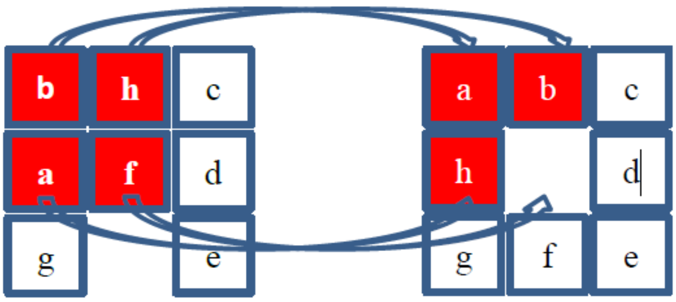
```
initial_state(Init),
findall([Action,Suiv],  rule(Action,_,Init,Suiv),Suiv_Liste).
```

Développement des 2 heuristiques

Heuristique 1 : nombre de pièces mal placées

```
%***************
% HEURISTIQUE no 1
%***************
% Nombre de pieces mal placees dans l'etat courant U
% par rapport a l'etat final F


% Suggestions : définir d'abord le prédicat coordonnees(Piece,Etat,Lig,Col) qui associe
% à une pièce présente dans Etat ses coordonnees (Lig= numero de ligne, Col= numero de Colonne)

coordonnees([L,C], Mat, Elt) :- nth1(L,Mat,Ligne), nth1(C,Ligne,Elt).

% Definir ensuite le predicat malplace(P,U,F) qui est vrai si les coordonnes de P dans U
% et dans F sont differentes.
% On peut également comparer les pieces qui se trouvent aux mêmes coordonnees dans U et
% dans H et voir s'il sagit de la même piece.

malplace(P,U,F) :- coordonnees([L,C],U,P), \+coordonnees([L,C],F,P), \+P = vide.

% Definir enfin l'heuristique qui détermine toutes les pièces mal placées (voir prédicat findall)
% et les compte (voir prédicat length)

heuristique1(U, H) :-
    final_state(Fin),
    findall(P,malplace(P,U,Fin),List),
    length(List, H).
```

Figure 3 : Heuristique 1



situation initiale U_0 situation finale F

$$h_1(U_0) = |\{b, h, a, f\}| = 4$$

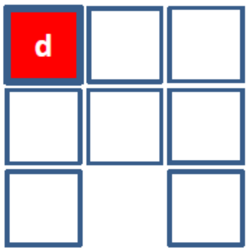
Figure 4 : Enoncé heuristique 1

Nous trouvons bien 4 lorsque nous appliquons l'heuristique 1 sur l'état initial.

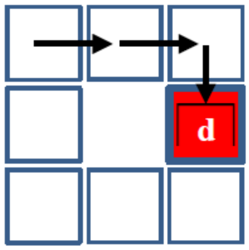
Heuristique 2 : distance de Manhattan

```
%****************
% HEURISTIQUE no 2
%****************

% Somme des distances de Manhattan à parcourir par chaque piece
% entre sa position courante et sa positon dans l'etat final

distance_manhattan(U,F,P,H) :-
    coordonnees([L1,C1],U,P),
    coordonnees([L2,C2],F,P),
    H is abs(L1-L2)+abs(C1-C2).

heuristique2(U, H) :-
    final_state(Fin),
    findall(X,(distance_manhattan(U,Fin,P,X),P\=vide),List),
    sum_list(List,H).
```

Figure 5 : Heuristique 2



Une situation U



La situation finale F

$$DM(d,U,F) = 3$$

La requête suivante :

```
distance_manhattan([

[d,vide,vide],
[vide,vide,vide],
[vide,vide,vide]],

[[vide,vide,vide],
[vide,vide,d],
[vide,vide,vide]],d,H).
```

Nous donne bien H=3. (Nous plaçons arbitrairement « vide » dans les cases restantes pour que ce ne soit pas des « d »)

Implémentation de A*

Prédicat main/0

```
%***********************************************
% Main
%***********************************************

main :-
    % état initial
    initial_state(S0),
    % Calcul de H0, G0 = 0 et F0 = H0
    heuristique(S0,H0),
    G0 is 0,
    F0 is H0+G0,
    % initialisations Pf, Pu et Q
    % 3 AVLs vides
    empty(Pf),
    empty(Pu),
    empty(Q),
    % Insertion de [ [F0,H0,G0], S0 ] dans Pf
    % et de [S0, [F0,H0,G0], nil, nil] dans Pu
    insert([[F0,H0,G0],S0],Pf,Pf2),
    insert([S0,[F0,H0,G0],nil,nil],Pu,Pu2),
    % lancement de Aetoile
    aetoile(Pf2,Pu2,Q).
```

Figure 6 : Prédicat main/0

Prédicat aetoile/3

```
%*********************************************************************
% Aetoile
%*********************************************************************

% Cas Trivial 1 : si Pf et Pu sont vides, il n'y a aucun état pouvant
% être développé donc pas de solution au problème

aetoile(nil,nil,_) :-
    write("PAS DE SOLUTION : L'ÉTAT FINAL N'EST PAS ATTEIGNABLE.").

% Cas Trivial 2 : si le noeud de valeur F minimum de Pf correspond
% à la situation terminale, alors on a trouvé une solution
% et on peut l'afficher (prédicat affiche_solution)

aetoile(Pf,Pu,Q) :-
    suppress_min([_,U],Pf,_),
    final_state(U),
    affiche_solution(U,Pu,Q).

% Cas Général

aetoile(Pf,Pu,Q) :-
    % on enlève le noeud de Pf correspondant à l'état U
    % à développer (celui de valeur F minimale)
    % et on enlève aussi le noeud frère associé dans Pu
    suppress_min([[F,H,G],U],Pf,Pf2),
    not(final_state(U)),
    suppress([U,[F,H,G],Pere,Action],Pu,Pu2),
    % développement de U
    expand(U,Lsucc,G),
    loop_successors(Lsucc,Pf2,Pu2,Q,Pf_new,Pu_new),
    % U ayant été développé et supprimé de P, il reste à l'insérer
    % le noeud [U,Val,…,..] dans Q,
    insert([U,[F,H,G],Pere,Action],Q,Q_new),
    % Appeler récursivement aetoile avec les nouveaux ensembles
    % Pf_new, Pu_new et Q_new
    aetoile(Pf_new,Pu_new,Q_new).
```

Figure 7 : Prédicat aetoile/3

Prédicat affiche_solution/3

```
%*********************************
% Affiche_solution
%*********************************

affiche_solution(nil,_,_).

affiche_solution(U,Pu,Q):-
    U \= nil,
    belongs([U,_,Pere,Action],Q),
    affiche_solution(Pere,Pu,Q),
    write(Action),
    write('->'),
    write_state(U),
    writeln(" ").

affiche_solution(U,Pu,Q):-
    U \= nil,
    belongs([U,_,Pere,Action],Pu),
    affiche_solution(Pere,Pu,Q),
    write(Action),
    write('->'),
    write_state(U),
    writeln(" ").
```

Figure 8 : Prédicat affiche_solution/3

Prédicat expand/3

```
%*************************************************************
% Expand
%*************************************************************

expand(U,Lsucc,G):-
    % déterminer tous les noeuds contenant un état
    % successeur S de la situation U et calculer leur
    % évaluation [Fs, Hs, Gs] connaissant Gu et le coût
    % pour passer de U à S.
    findall([Succ,[Fsucc,Gsucc,Hsucc],U,Action],
        (rule(Action,Cout,U,Succ),
            Gsucc is G+Cout,
            heuristique(Succ,Hsucc),
            Fsucc is Gsucc+Hsucc),
        Lsucc).
```

Figure 9 : Prédicat expand/3

Prédicat loop_successors/6

```
%*************************************************************
% Loop_successors
%*************************************************************

% Cas trivial : terminaison
loop_successors([],Pf,Pu,_,Pf,Pu).

% traiter chaque noeud successeur (prédicat loop_successors) :

% - si S est connu dans Q alors oublier cet état
% (S a déjà été développé)
loop_successors([[Succ,_,_,_]|Rest],Pf,Pu,Qs,Pf_new,Pu_new):-
    belongs([Succ,_,_,_],Qs),
    loop_successors(Rest,Pf,Pu,Qs,Pf_new,Pu_new).

% - si S est connu dans Pu alors garder le terme associé
% à la meilleure évaluation (dans Pu et dans Pf)
%    * F<Fsucc, on continue sans rien modifier
loop_successors([[Succ,[Fsucc,_,_],_,_]|Rest],
                Pf,Pu,Qs,Pf_new,Pu_new):-
    not(belongs([Succ,_,_,_],Qs)),
    belongs([Succ,[F,_,_],_,_],Pu),
    F<Fsucc,
    loop_successors(Rest,Pf,Pu,Qs,Pf_new,Pu_new).
%    * F>Fsucc, on a trouvé un meilleur chemin :
%    on modifie Pu et Pf
loop_successors([[Succ,[Fsucc,Hsucc,Gsucc],Pere,Action]|Rest],
                                Pf,Pu,Qs,Pf_new,Pu_new):-
    not(belongs([Succ,_,_,_],Qs)),
    belongs([Succ,[F,H,G],_,_],Pu),
    F>Fsucc,
    suppress([Succ,[F,H,G],_,_],Pu,Pu2),
    insert([Succ,[Fsucc,Hsucc,Gsucc],Pere,Action],Pu2,Pu3),
    suppress([[F,H,G],Succ],Pf,Pf2),
    insert([[Fsucc,Hsucc,Gsucc],Succ],Pf2,Pf3),
    loop_successors(Rest,Pf3,Pu3,Qs,Pf_new,Pu_new).

% - sinon (S est une situation nouvelle) il faut créer
% un nouveau terme à insérer dans Pu (idem dans Pf)
loop_successors([[Succ,[Fsucc,Hsucc,Gsucc],Pere,Action]|Rest],
                                Pf,Pu,Qs,Pf_new,Pu_new):-
    not(belongs([Succ,_,_,_],Qs)),
    not(belongs([Succ,_,_,_],Pu)),
    insert([Succ,[Fsucc,Hsucc,Gsucc],Pere,Action],Pu,Pu2),
    insert([[Fsucc,Hsucc,Gsucc],Succ],Pf,Pf2),
    loop_successors(Rest,Pf2,Pu2,Qs,Pf_new,Pu_new).
```

Figure 10 : Prédicat loop_successors/6

Analyse Expérimentale

Résultat de l'algorithme, taquin 3x3

```
?- main.
nil->[b,h,c]
[a,f,d]
[g,vide,e]

up->[b,h,c]
[a,vide,d]
[g,f,e]

up->[b,vide,c]
[a,h,d]
[g,f,e]

left->[vide,b,c]
[a,h,d]
[g,f,e]

down->[a,b,c]
[vide,h,d]
[g,f,e]

right->[a,b,c]
[h,vide,d]
[g,f,e]

true
```

*Figure 11 : Résultats A**

Le programme retourne le résultat ci-contre avec pour état initial U0 et état final F définis dans l'introduction.

A* trouve-t-il la solution pour la situation initiale suivante ?

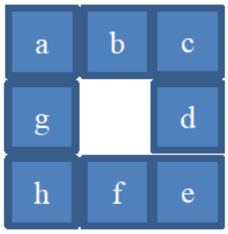


Figure 12 : A autre situation initiale*

Dans la situation ci-dessus, l'algorithme A* retourne « false ». En effet, il n'existe pas de solution à partir de cet état (non connexe à l'état final).

Temps de calcul de A* et influence du choix de l'heuristique

Heuristique	Taille de séquences optimales	Temps de calcul (ms)
N°1	6	68
N°2	6	52

Adaptation : Rubik's Cube

Pour le Rubik's Cube, nous aurions une liste avec les 6 faces du cubes (chacune représentée comme dans le Taquin 3x3) et 18 actions possibles : en prenant une orientation fixe, 2 sens de rotation * 3 lignes ou colonnes * 3 axes.

TP2 – Algo minmax – Application au TicTacToe

Familiarisation avec le problème du TicTacToe 3x3

Interprétation des requêtes suivantes :

- `situation_initiale(S), joueur_initial(J).`

  Cette requête retourne la situation initiale S (toutes les cases vides) et le joueur qui commence J (« x » par convention)

- `situation_initiale(S), nth1(3,S,Lig), nth1(2,Lig,o).`

  Cette requête retourne la situation initiale S à laquelle on rajoute un « o » en ligne 3 colonne 2.

Prédicat situation_terminale/2

```
/************************************************
DEFINIR ICI a l'aide du predicat ground/1 comment
reconnaitre une situation terminale dans laquelle il
n'y a aucun emplacement libre : aucun joueur ne peut
continuer a jouer (quel qu'il soit).
************************************************/

situation_terminale(_Joueur, Situation) :- ground(Situation).
```

Figure 13 : Prédicat situation_terminale/2

Prédicat alignement/2 : ligne, colonne ou diagonale

```
/***************************
DEFINITIONS D'UN ALIGNEMENT
***************************/

alignement(L, Matrix) :- ligne(    L,Matrix).
alignement(C, Matrix) :- colonne(  C,Matrix).
alignement(D, Matrix) :- diagonale(D,Matrix).
```

Figure 14 : Prédicat alignement/2

Un alignement est soit une ligne, soit une colonne soit une diagonale : nous détaillons donc chaque cas.

```
ligne(L, M) :- nth1(_N,M,L).

nthcolonne(_N,[],[]).
nthcolonne(N,[Elem|CReste],[Ligne|LReste]):-
    nth1(N,Ligne,Elem),
    nthcolonne(N,CReste,LReste).

colonne(C,M) :- nthcolonne(_,C,M).
```

Figure 15 : Prédicats ligne/2 et colonne/2

```
diagonale(D, M) :-
    premiere_diag(1,D,M).

diagonale(D, M) :-
    length(M,N),
    seconde_diag(N,D,M).

premiere_diag(_,[],[]).
premiere_diag(K,[E|D],[Ligne|M]) :-
    nth1(K,Ligne,E),
    K1 is K+1,
    premiere_diag(K1,D,M).

seconde_diag(_,[],[]).
seconde_diag(K,[E|D],[Ligne|M]) :-
    nth1(K,Ligne,E),
    K1 is K-1,
    seconde_diag(K1,D,M).
```

Figure 16 : Prédicat diagonale/2

Il y a 2 sortes de diagonales dans une matrice carrée :

- la premiere diagonale (principale) : (A I)
- la seconde diagonale : (Z R)

```
A . . . . . . . Z
. \ . . . . . / .
. . \ . . . / . .
. . . \ . / . . .
. . . . X . . .
. . . / . \ . . .
. . / . . . \ . .
. / . . . . . \ .
R . . . . . . . I
```

Figure 17 : Exemple matrice carrée

Fonctionnement du prédicat alignement/2 sur la matrice :

```
a . b . c
d . e . f
g . h . i
```

Figure 18 : Exemple matrice pour alignement/2

Alignement	Type d'alignement
[a,b,c]	Ligne
[d,e,f]	Ligne
[g,h,i]	Ligne
[a,d,g]	Colonne
[b,e,h]	Colonne
[c,f,i]	Colonne
[a,e,i]	Première diagonale
[c,e,g]	Seconde diagonale

Prédicat possible/2 et unifiable/2

```
/*****************************
DEFINITION D'UN ALIGNEMENT
POSSIBLE POUR UN JOUEUR DONNE
*****************************/

possible([X|L], J) :- unifiable(X,J), possible(L,J).
possible(  [],   _).

    /* Attention
    il faut juste verifier le caractere unifiable
    de chaque emplacement de la liste, mais il ne
    faut pas realiser l'unification.
    */

unifiable(X,_J) :- var(X).
unifiable(X,J) :- ground(X),X=J
```

Le prédicat possible/2 retourne vrai si le joueur J donné peut encore gagner avec un alignement donné

Figure 19 : Prédicat possible/2 et unifiable/2

Tests unitaires et résultats :

```
possible([x,x,x],x).
true.
possible([_,_,_],x).
true.
possible([x,_,x],x).
true.
possible([o,_,x],x).
false.
possible([o,_,_],x).
false.
```

Prédicats alignement_gagnant/2 et alignement_perdant/2

```
/***********************************
DEFINITION D'UN ALIGNEMENT GAGNANT
OU PERDANT POUR UN JOUEUR DONNE J
***********************************/
/*
Un alignement gagnant pour J est un alignement
possible pour J qui n'a aucun element encore libre.
*/

/*
Remarque : le predicat ground(X) permet de verifier qu'un terme
prolog quelconque ne contient aucune partie variable (libre).
*/

/* Un alignement perdant pour J est un alignement gagnant pour son adversaire. */

alignement_gagnant(Ali, J) :- ground(Ali), possible(Ali,J).

alignement_perdant(Ali, J) :- adversaire(J,J2), alignement_gagnant(Ali,J2).
```

Figure 20 : Prédicats alignement_gagnant/2 et alignement_perdant/2

Tests unitaires et résultats :

```
alignement_gagnant([x,x,x],x).
true.
alignement_gagnant([_,_,_],x).
false.
alignement_gagnant([x,_,x],x).
false.
alignement_gagnant([o,_,x],x).
false.
alignement_gagnant([o,_,_],x).
false.
```

```
alignement_perdant([x,x,x],o).
true.
alignement_perdant([o,o,o],x).
true.
alignement_perdant([_,o,o],x).
false.
```

Développement de l'heuristique

```
/* ******************************
DEFINITION D'UN ETAT SUCCESSEUR
****************************** */

/*
Il faut definir quelle operation subit la matrice
M representant l'Etat courant
lorsqu'un joueur J joue en coordonnees [L,C]
*/

successeur(J,Etat,[L,C]) :-
    nth1(L,Etat,Ligne),
    nth1(C,Ligne,J).
```

Figure 22 : Prédicat successeur/3

```
/**************************************
EVALUATION HEURISTIQUE D'UNE SITUATION
**************************************/

/*
1/ l'heuristique est +infini si la situation J est gagnante pour J
2/ l'heuristique est -infini si la situation J est perdante pour J
3/ sinon, on fait la difference entre :
   le nombre d'alignements possibles pour J
moins
   le nombre d'alignements possibles pour l'adversaire de J
*/

heuristique(J,Situation,H) :-       % cas 1
   H = 10000,                  % grand nombre approximant +infini
   alignement(Alig,Situation),
   alignement_gagnant(Alig,J), !.

heuristique(J,Situation,H) :-       % cas 2
   H = -10000,                 % grand nombre approximant -infini
   alignement(Alig,Situation),
   alignement_perdant(Alig,J), !.

    % on ne vient ici que si les cut precedents n'ont pas fonctionne,
    % c-a-d si Situation n'est ni perdante ni gagnante.

heuristique(J,Situation,H) :-       % cas 3
    findall(Ali,(alignement(Ali,Situation),possible(Ali,J)),ListJ),
    adversaire(J,J2),
    findall(Ali,(alignement(Ali,Situation),possible(Ali,J2)),ListJ2),
    length(ListJ,H1),
    length(ListJ2,H2),
    H is H1-H2.
```

Figure 21 : Prédicat heuristique/3

Tests unitaires et résultats :

```
heuristique(x,[[_,_,_],[x,x,x],[_,_,_]],H).
H = 10000.
heuristique(x,[[_,_,_],[_,_,x],[_,_,_]],H).
H = 2.
heuristique(x,[[_,_,_],[_,x,_],[_,_,_]],H).
H = 4.
heuristique(o,[[_,_,_],[_,x,_],[_,_,_]],H).
H = -4.
```

Développement de l'algorithme Negamax

Quel prédicat permet de connaître sous forme de liste l'ensemble des couples [Coord, Situation_Resultante]?

```
successeurs(J,Etat,Succ).
```

« Succ » est la liste des couples [Coord, Situation_Resultante].

Joueur « x », Situation initiale :

```
Succ = [
[[1,1],[[x,_,_],[_,_,_],[_,_,_]]],
[[1,2],[[_,x,_],[_,_,_],[_,_,_]]],
[[1,3],[[_,_,x],[_,_,_],[_,_,_]]],
[[2,1],[[_,_,_],[x,_,_],[_,_,_]]],
[[2,2],[[_,_,_],[_,x,_],[_,_,_]]],
[[2,3],[[_,_,_],[_,_,x],[_,_,_]]],
[[3,1],[[_,_,_],[_,_,_],[x,_,_]]],
[[3,2],[[_,_,_],[_,_,_],[_,x,_]]],
[[3,3],[[_,_,_],[_,_,_],[_,_,x]]]].
```

Prédicat negamax/5

negamax(+J, +Etat, +P, +Pmax, [?Coup, ?Val])

Retourne pour un joueur J donné, devant jouer dans une situation donnée Etat, de profondeur donnée P, le meilleur couple [Coup, Valeur] après une analyse aller jusqu'a la profondeur Pmax.

Il y a 3 cas a décrire (donc 3 clauses pour negamax/5)

1/ la profondeur maximale est atteinte : on ne peut pas développer cet Etat ; il n'y a donc pas de coup possible à jouer (Coup = rien) et l'évaluation de Etat est faite par l'heuristique.

```
% Cas 1 : la profondeur maximale est atteinte
negamax(J,Etat,Pmax,Pmax,[nil,Val]):-heuristique(J,Etat,Val).
```

Figure 23 : Prédicat negamax/5 cas 1

2/ la profondeur maximale n'est pas atteinte mais J ne peut pas jouer ; au TicTacToe un joueur ne peut pas jouer quand le tableau est complet (totalement instancie) ; il n'y a pas de coup à jouer (Coup = rien) et l'évaluation de Etat est faite par l'heuristique.

```
% Cas 2 : la profondeur maximale n'est pas  atteinte mais J ne peut pas jouer
negamax(J,Etat,P,Pmax,[_,Val]):-
     P < Pmax,
     situation_terminale(J,Etat),
     heuristique(J,Etat,Val).
```

Figure 24 : Prédicat negamax/5 cas 2

3/ la profondeur maxi n'est pas atteinte et J peut encore jouer. Il faut évaluer le sous-arbre complet issu de Etat ;

- on détermine d'abord la liste de tous les couples [Coup_possible, Situation_suivante] via le prédicat successeurs/3 (deja fourni, voir plus bas).

- cette liste est passée a un prédicat intermédiaire : loop_negamax/5, charge d'appliquer negamax sur chaque Situation_suivante ; loop_negamax/5 retourne une liste de couples [Coup_possible, Valeur]

- parmi cette liste, on garde le meilleur couple, c-a-d celui qui a la plus petite valeur (cf. prédicat meilleur/2); soit [C1,V1] ce couple optimal. Le prédicat meilleur/2 effectue cette sélection.

- finalement le couple retourne par negamax est [Coup, V2] avec : V2 is -V1 (cf. convention negamax vue en cours).

```
% Cas 3 : la profondeur maxi n'est pas atteinte et J peut encore jouer
negamax(J,Etat,P,Pmax,[Coup,Val]):-
    P < Pmax,
    successeurs(J,Etat,Succ),
    loop_negamax(J,P,Pmax,Succ,Liste_Couples),
    meilleur(Liste_Couples,[Coup,V1]),
    Val is -V1.
```

Figure 25 : Prédicat negamax/5 cas 3

Prédicat loop_negamax/5

Ce prédicat est une boucle permettant d'appliquer negamax à chaque situation suivante.

loop_negamax(+J,+P,+Pmax,+Successeurs,?Liste_Couples)

Retourne la liste des couples [Coup, Valeur_Situation_Suivante] à partir de la liste des couples [Coup, Situation_Suivante]

```
loop_negamax(_,_, _  ,[],                []).
loop_negamax(J,P,Pmax,[[Coup,Suiv]|Succ],[[Coup,Vsuiv]|Reste_Couples]) :-
    % On récupère le couple [Coup,Suiv] dans Successeurs et
    % on met le couple [Coup,Vsuiv] dans Liste_Couples
    loop_negamax(J,P,Pmax,Succ,Reste_Couples),
    % On alterne de joueur
    adversaire(J,A),
    % On actualise la profondeur (+1)
    Pnew is P+1,
    % On relance negamax avec le nouveau joueur, la nouvelle profondeur et l'état suivant
    % On met _ à la place du coup car on ne sait pas encore quel est le meilleur coup
    negamax(A,Suiv,Pnew,Pmax,[_,Vsuiv]).
```

Figure 26 : Prédicat loop_negamax/5

Prédicat meilleur/2

Ce prédicat permet la sélection du couple qui a la plus petite valeur V

meilleur(+Liste_de_Couples, ?Meilleur_Couple)

```
meilleur([],[]).
% Cas 1 : le meilleur dans une liste a un seul element est cet element
meilleur([Couple],Couple).
% Cas 2 : le meilleur dans une liste [X|L] avec L \= [], est obtenu en comparant
%      X et Y,le meilleur couple de L
%      Entre X et Y on garde celui qui a la petite valeur de V.
meilleur([[Coup,Vsuiv]|Reste_Couples],Meilleur_Couple):-
    Reste_Couples \= [],
    meilleur(Reste_Couples,[McoupSuiv,MVSuiv]),
    (Vsuiv < MVSuiv
        % Si X < Y, on garde X
        -> Meilleur_Couple=[Coup,Vsuiv]
        % Sinon, on garde Y
        ;  Meilleur_Couple=[McoupSuiv,MVSuiv]
    ).
```

Figure 27 : Prédicat meilleur/2

Test unitaire et résultat obtenu

```
meilleur([[a,-1],[b,-51],[c,-62],[d,-4]],[Mcoup,MV]).
Mcoup = c,
MV = -62 ;
```

Prédicat main/3

```
    /*****************
    PROGRAMME PRINCIPAL
    ******************/

main(B,V, Pmax) :-
    situation_initiale(Etat),
    joueur_initial(J),
    negamax(J,Etat,0,Pmax,[B,V]).
```

Figure 28 : Prédicat main/3

Expérimentation et extensions

Meilleur coup à jouer et gain espéré

Profondeur Max	Meilleur coup	Gain espéré
1	[2,2]	4
2	[2,2]	1
3	[2,2]	3
4	[2,2]	1
5	[2,2]	3
6	[2,2]	1
7	[2,2]	2
8	?	?
9	?	?

Situations symétriques de situations déjà développées

Pour ne plus développer inutilement des situations symétriques de situations déjà développées, nous pourrions établir un prédicat *situation_symetrique(situation1, situation2)* qui retournerai vrai si la situation2 est obtenable par rotation (et/ou symétrie) de la situation1. Ainsi avant de développer une situation, nous pourrions vérifier si une situation symétrique a déjà été développée ou non .

```
    /******************
    SITUATION SYMETRIQUE
    *******************/

matrix_rotated(Xss, Zss) :-
    transpose(Xss, Yss),
    maplist(reverse, Yss, Zss).

nrotate(0,M,M).

nrotate(N,M,M2) :-
    not(N = 0),
    N1 is N-1,
    matrix_rotated(M,M3),
    nrotate(N1,M3,M2).

situation_symetrique(S1,S2) :- % Rotation
    between(1,3,N), % 4eme rotation = lui-meme
    nrotate(N,S1,S2).

situation_symetrique(S1,S2) :- % Symetrie horizontale + rotation
    reverse(S1,S3),
    between(1,3,N), % 4eme rotation = lui-meme
    nrotate(N,S3,S2).
```

Figure 29 : Prédicat situation_symetrique/2

Adaptation : Jeu du puissance 4

Dans le cas du jeu du puissance 4, nous pourrions adapter notre programme en changeant les coups possibles, ainsi que les alignements. Il faudrait prendre en compte le fait que l’on ne peut poser des pions « qu’en bas de la grille » (tout en bas ou au-dessus d’autre pion déjà placé). Les alignements quant à eux seraient des lignes, des colonnes ou des diagonales de 4 pions d’une même couleur. Il faudrait également l’utilisation du programme sur des grilles dont au moins une des dimensions (nombre de lignes ou de colonnes) est > 4.

Amélioration : recherche Alpha-Beta

Pour améliorer notre algorithme en élaguant certains coups inutiles à l'aide de la recherche Alpha-Beta, il faudrait rajouter des arguments « alpha » et « beta » dans les prédicats *loop_negamax* et *negamax*, afin de les actualiser lors du parcours et d'élaguer certaines explorations en fonction de ces valeurs.

INSA Toulouse
135, avenue de Rangueil
31077 Toulouse Cedex 4 - France
www.insa-toulouse.fr

Université Fédérale Toulouse Midi-Pyrénées

Liberté • Égalité • Fraternité
RÉPUBLIQUE FRANÇAISE

MINISTÈRE
DE L'ÉDUCATION NATIONALE,
DE L'ENSEIGNEMENT SUPÉRIEUR
ET DE LA RECHERCHE