

**Compte rendu de TP**

**Intelligence Artificielle**

15 mars 2021

HOK Jean-Rémy

INSA\* Promotion 55, 4ème Année IR\*

135, Avenue de Rangueil

31077 Toulouse Cedex 4

\*INSA : Institut National des Sciences Appliquées

\*IR : Informatique et Réseaux

**Compte rendu de TP**

**Intelligence Artificielle**

15 mars 2021

HOK Jean-Rémy

INSA\* Promotion 55, 4ème Année IR\*

**INSA Toulouse**

135, Avenue de Rangueil

31077 Toulouse Cedex 4

\*INSA : Institut National des Sciences Appliquées

\*IR : Informatique et Réseaux

Table des matières

[Introduction 1](#_Toc66887073)

[TP1 – Algorithme A\* – Application au Taquin 2](#_Toc66887074)

[Familiarisation avec le problème du Taquin 3x3 2](#_Toc66887075)

[Développement des 2 heuristiques 3](#_Toc66887076)

[Heuristique 1 : nombre de pièces mal placées 3](#_Toc66887077)

[Heuristique 2 : distance de Manhattan 4](#_Toc66887078)

[Implémentation de A\* 5](#_Toc66887079)

[Analyse Expérimentale 9](#_Toc66887080)

[Résultat de l’algorithme, taquin 3x3 9](#_Toc66887081)

[Temps de calcul de A\* et influence du choix de l’heuristique 9](#_Toc66887082)

[Adaptation  : Rubik’s Cube 9](#_Toc66887083)

[TP2 – Algo minmax – Application au TicTacToe 10](#_Toc66887084)

[Familiarisation avec le problème du TicTacToe 3x3 10](#_Toc66887085)

[Développement de l’heuristique 13](#_Toc66887086)

[Développement de l’algorithme Negamax 14](#_Toc66887087)

[Expérimentation et extensions 17](#_Toc66887088)

[Meilleur coup à jouer et gain espéré 17](#_Toc66887089)

[Situations symétriques de situations déjà développées 17](#_Toc66887090)

[Adaptation : Jeu du puissance 4 17](#_Toc66887091)

[Amélioration : recherche Alpha-Beta 18](#_Toc66887092)

# 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

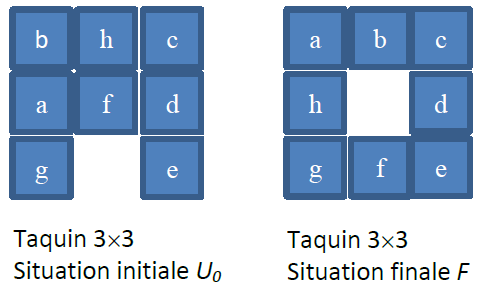


Figure 1 : Contexte Taquin

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

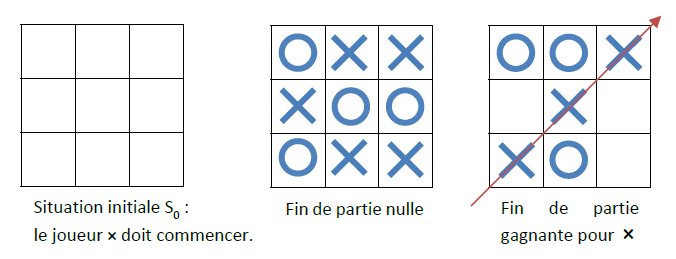


Figure 2 : Contexte TicTacToe

# 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

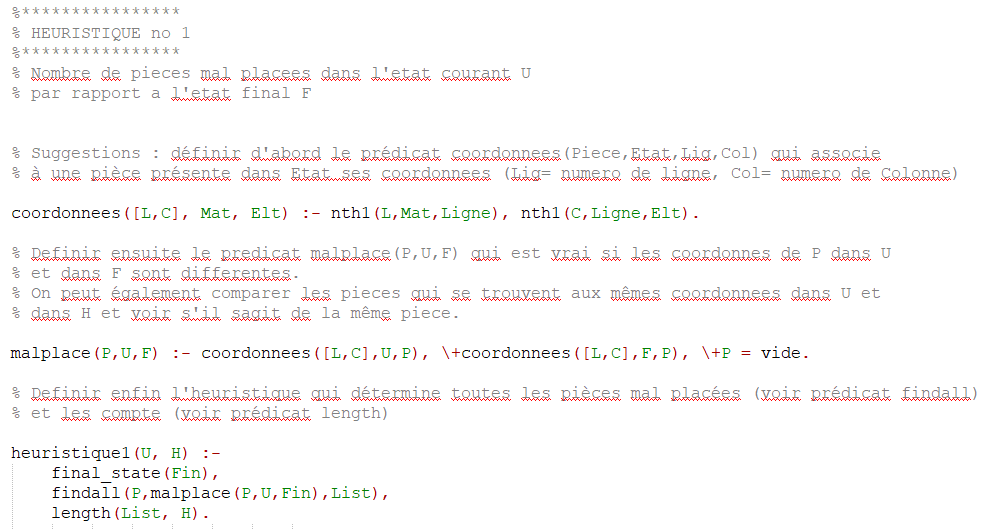


Figure 3 : Heuristique 1

|  |  |
| --- | --- |
| Figure 4 : Enoncé heuristique 1 | Nous trouvons bien 4 lorsque nous appliquons l’heuristique 1 sur l’état initial. |

### Heuristique 2 : distance de Manhattan

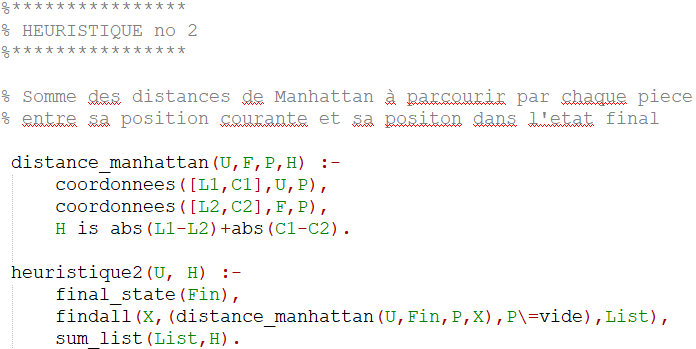


Figure 5 : Heuristique 2

|  |  |
| --- | --- |
|  | 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

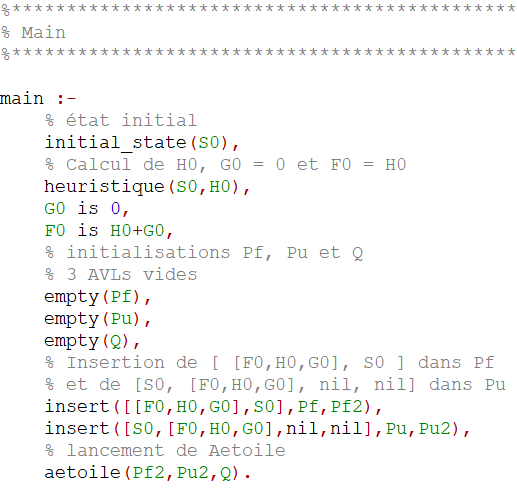


Figure 6 : Prédicat main/0

#### Prédicat aetoile/3

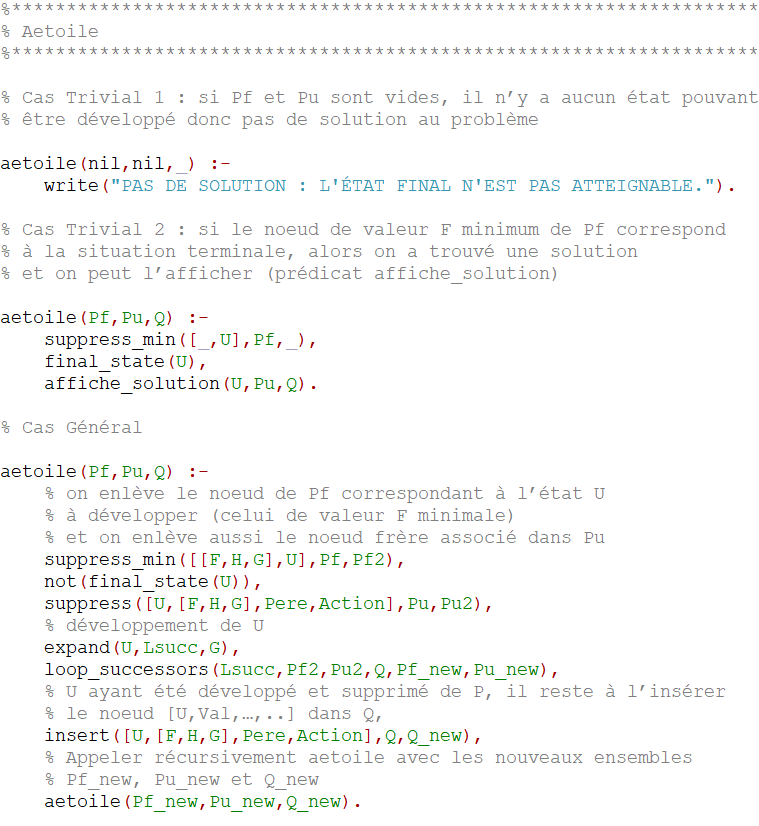


Figure 7 : Prédicat aetoile/3

#### Prédicat affiche\_solution/3

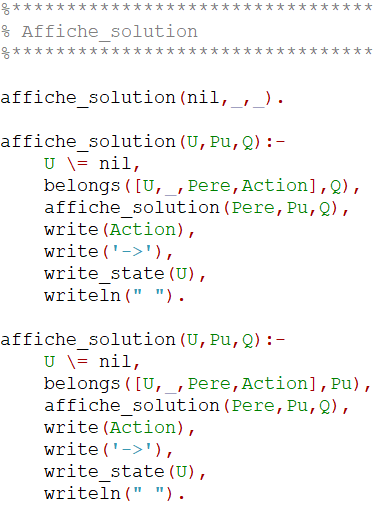


Figure 8 : Prédicat affiche\_solution/3

#### Prédicat expand/3

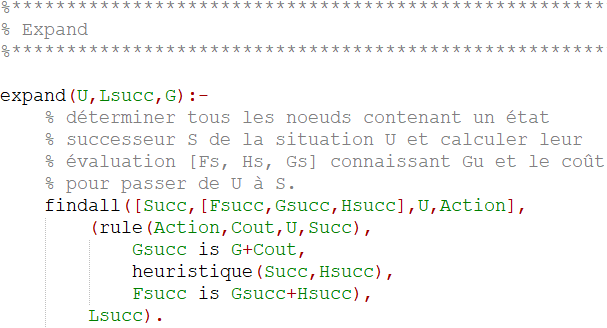


Figure 9 : Prédicat expand/3

#### Prédicat loop\_successors/6

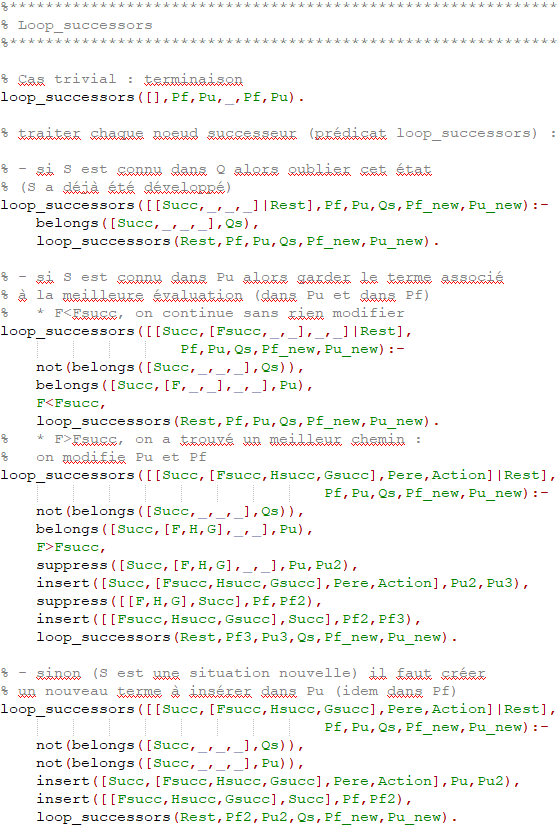


Figure 10 : Prédicat loop\_successors/6

## Analyse Expérimentale

### Résultat de l’algorithme, taquin 3x3

|  |  |
| --- | --- |
| Figure 11 : Résultats A\* | Le programme retourne le résultat ci-contre avec pour état initial U0 et état final F définis dans [l’introduction](#_Introduction).    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

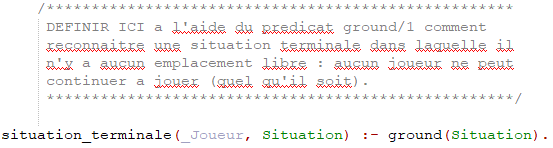


Figure 13 : Prédicat situation\_terminale/2

#### Prédicat alignement/2 : ligne, colonne ou diagonale

|  |  |
| --- | --- |
| Figure 14 : Prédicat alignement/2 | Un alignement est soit une ligne, soit une colonne soit une diagonale : nous détaillons donc chaque cas. |

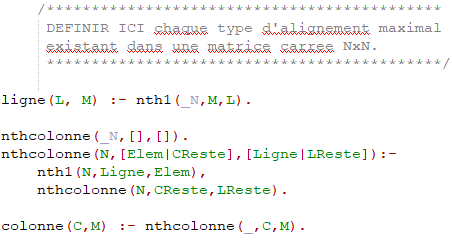


Figure 15 : Prédicats ligne/2 et colonne/2

|  |  |
| --- | --- |
| 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)  Figure 17 : Exemple matrice carrée |

Fonctionnement du prédicat alignement/2 sur la matrice :



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

|  |  |
| --- | --- |
| Figure 19 : Prédicat possible/2 et unifiable/2 | Le prédicat possible/2 retourne vrai si le joueur J donné peut encore gagner avec un alignement donné |

|  |  |
| --- | --- |
| 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

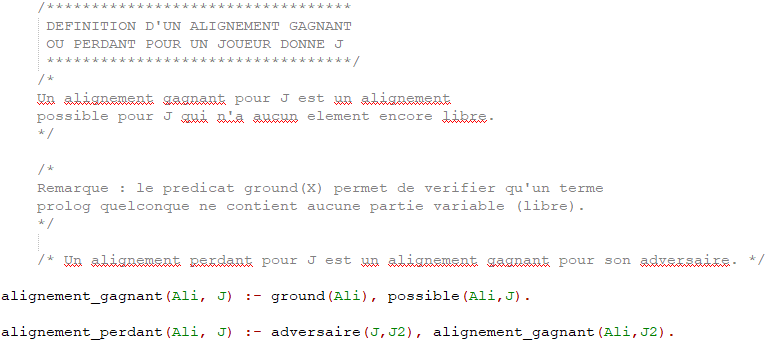


Figure 20 : Prédicats alignement\_gagnant/2 et alignement\_perdant/2

|  |  |  |
| --- | --- | --- |
| 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

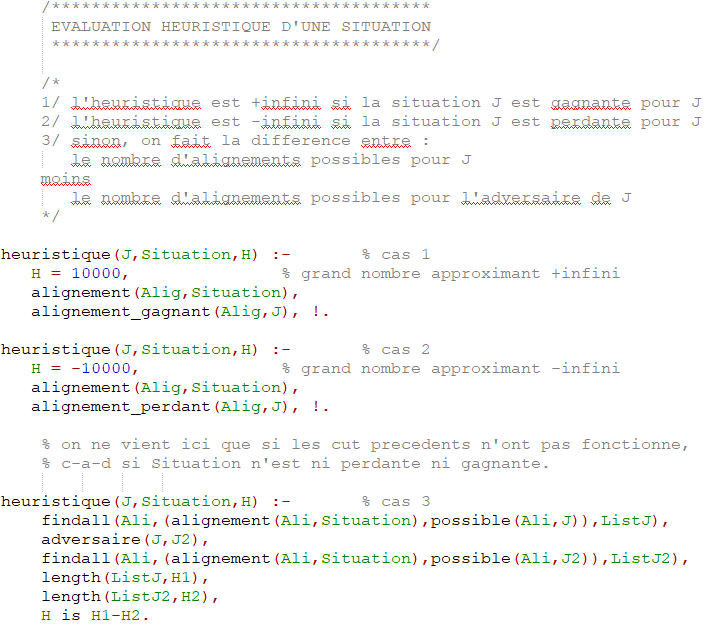


Figure 21 : Prédicat heuristique/3

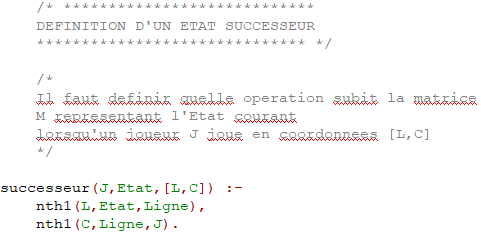


Figure 22 : Prédicat successeur/3

|  |  |
| --- | --- |
| 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.



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.

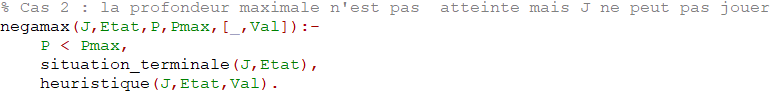


Figure 24 : Prédicat negamax/5 cas 2

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).

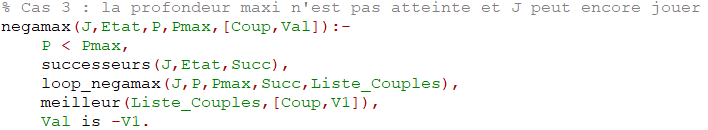


Figure 25 : Prédicat negamax/5 cas 3

#### 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]

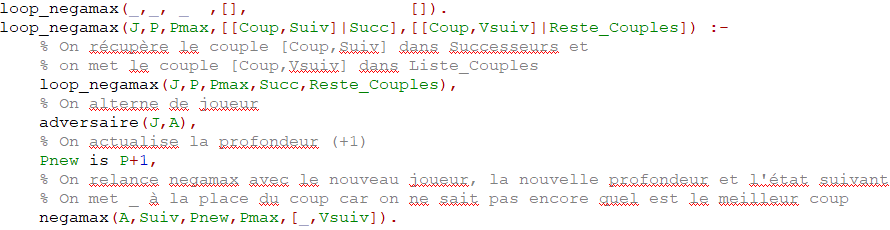


Figure 26 : Prédicat loop\_negamax/5

#### Prédicat meilleur/2

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

meilleur(+Liste\_de\_Couples, ?Meilleur\_Couple)

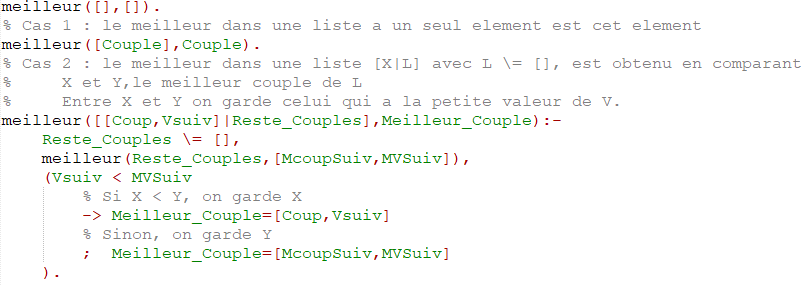


Figure 27 : Prédicat meilleur/2

|  |  |
| --- | --- |
| Test unitaire et résultat obtenu | meilleur([[a,-1],[b,-51],[c,-62],[d,-4]],[Mcoup,MV]).  Mcoup = c,  MV = -62 ; |

#### Prédicat main/3

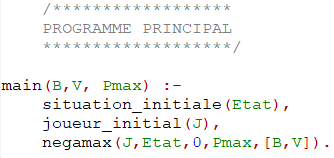


Figure 28 : Prédicat main/3

## 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 . | 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.

