BARNABE Elise – FAURE Paul 4IR A1

TP Intelligence Artificielle – Compte Rendu

TP1 Algorithme A* Application au Taquin

Familiarisation avec le problème du taquin 3x3

a) 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]]).

b) A quelles questions permettent de répondre les requêtes suivantes :

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

Nth1(Index, List, Elmt) est vrai quand Elmt est le Index ième élément de List.
Ici, grâce au premier nth1 on va prendre tour à tour toutes les lignes de Ini stockées dans Ligne et leur numéro dans L puis grâce au deuxième on regarde s'il y a d dans cette ligne et si oui on retourne son numéro de colonne dans C. Cela permet de connaître les coordonnées de d dans Ini.

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

Ici c'est le même principe mais dans le premier nth1 on récupère dans Ligne le contenu de la troisième ligne puis grâce au deuxième nth1 on récupère dans P le deuxième élément de Ligne. Donc cette ligne permet de voir ce qu'il y a aux coordonnées [3,2].

Nous voyons donc grâce à ces deux questions l'importance des variables libres et liées dans une même question Prolog. Si :

- les coordonnées sont liées et l'élément libre : on obtient l'élément présent à ces coordonnées
- l'élément est lié et les coordonnées libres : on obtient les coordonnées de l'élément
- les deux sont liés : on vérifie que cet élément est bien à ces coordonnées

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

wellplaced(Lettre) :- initial_state(Ini), nth1(L, Ini, Ligne), nth1(C, Ligne, Lettre), final_state(F), nth1(L, Fin, Ligne2), nth1(C, Ligne2, Lettre).

Où Lettre est la lettre de la pièce donnée.

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

initial_state(Ini), rule(down, _ , Ini, S).

On bouge vers le bas. Le résultat est stocké dans S.

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

initial_state(Ini), findall(S, rule(_, _, Ini, S), List2).

f) 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(Ini), findall([A, S], rule(A, _ , Ini, S), List2).

Développement des 2 heuristiques

a) Définition du prédicat coordonnées

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

Ce prédicat relie des coordonnées [Ligne, Colonne] à un élément d'une matrice. Pour cela on utilise le prédicat nth1.

b) Première heuristique : nombre de pièces mal placées

```
mal_place(Lettre, S, F) :-
  coordonnees(Lettre, S, [L,C]),
  coordonnees(Lettre, F, [L2,C2]),
  [L,C]\=[L2,C2],
  Lettre\=vide.

heuristique1(U, H) :-
  final_state(F),
  findall(X, mal_place(X, U, F), L),
  length(L, H).
```

Le prédicat mal_place(Lettre, S, F) renvoie vrai si les coordonnées de Lettre dans S sont différentes des coordonnées de Lettre dans F.
Enfin, dans l'heuristique, on cherche dans U, toutes les pièces mal placées en comparant toutes les pièces avec un état final. La valeur de l'heuristique est la taille de la liste des pièces mal placées, cela représente donc le nombre de pièces mal placées de l'état U.

c) Deuxième heuristique : distance de Manhattan

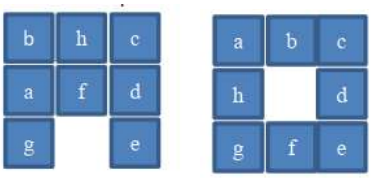
```
formule_manhattan(Lettre, U, F, D) :-
    coordonnees(Lettre, U, [L, C]),
    coordonnees(Lettre, F, [L2, C2]),
    X is abs(L-L2),
    Y is abs(C-C2),
    D is (X+Y).

 heuristique2(U, H) :-
    final_state(F),
    findall(D, (formule_manhattan(X, U, F, D), X\=vide), List),
    sumlist(List, H).
```

On définit d'abord le prédicat formule_manhattan(Lettre, U, F, D). Il calcule la distance minimale à parcourir pour la pièce Lettre dans l'état U pour rejoindre sa position dans l'état F. La formule est la suivante :

$$DistanceManhattan(Lettre, U, F) = |X_F - X_U| + |Y_F - Y_U|$$

Enfin, pour l'heuristique, on cherche la distance de Manhattan pour chaque pièce entre l'état U et l'état F, qui est un état final. Enfin, la valeur de l'heuristique est la somme des distances de Manhattan de toutes les pièces.

d) Tests avec des situations extrêmes



Taquin 3×3
Situation initiale U_0

Taquin 3×3
Situation finale F

Quand on compare la situation initiale U0 avec la situation finale F, on compte 4 pièces mal placées (b, h, a et f). Si on teste l'heuristique 1 sur U0 on doit donc obtenir 4.

```
initial_state(U0), heuristique1(U0, H).

H = 4,
U0 = [[b, h, c], [a, f, d], [g, vide, e]]
```

Le résultat obtenu est cohérent. Considérons l'heuristique 2 sur la situation initiale U0. Voici un tableau des distances de Manhattan pour chaque pièce de l'état U0 :

Pièce	a	b	c	d	e	f	g	h	Total
Distance de Manhattan	1	1	0	0	0	1	0	2	5

```
initial_state(U0), heuristique2(U0, H).

H = 5,
U0 = [[b, h, c], [a, f, d], [g, vide, e]]
```

Le résultat est conforme avec le calcul théorique.
Si on teste l'heuristique 1 sur la situation finale on doit obtenir 0 car aucune pièce n'est mal placée.

```
final_state(F), heuristique1(F, H).

F = [[a, b, c], [h, vide, d], [g, f, e]],
H = 0
```

De même pour l'heuristique 2, on doit aussi obtenir 0 car les deux états comparés pour les coordonnées sont les mêmes.

```
final_state(F), heuristique2(F, H).

F = [[a, b, c], [h, vide, d], [g, f, e]],
H = 0
```

Implémentation de A*

a) Développement du prédicat main

```
main :-
    initial_state(S0),
    heuristique(S0, H0),
    G0 is 0,
    F0 is H0+G0,
    empty(Pf),
    empty(Pu),
    empty(Q),
    insert([[F0,H0,G0], S0], Pf, Pf2),
    insert([S0, [F0,H0,G0], nil, nil] , Pu, Pu2),
    aetoile(Pf2, Pu2, Q).
```

On met dans S0 l'état initial, ensuite on calcule dans H0 l'heuristique correspondant à l'état S0. On fixe G0 (G est le coût d'un coup) à 0 et F0 à G0+H0. Il faut ensuite créer PF, Pu et Q qui sont trois AVL initialement vides. On insère dans Pf le nœud [[F0, H0, G0], S0] et dans Pu le nœud [S0, [F0, H0, G0], nil, nil]. Enfin, on fait une récursion du prédicat avec les nouveaux avl Pf, Pu et Q obtenus.

b) Développement du prédicat aetoile

```
aetoile(nil, nil, _) :-
    write("PAS DE SOLUTION : l'etat final n est pas atteignable").
aetoile(Pf, Ps , Qs) :-
    suppress_min([_, U], Pf, _),
    final_state(U),
    affiche_solution(Qs, Ps, U).
aetoile(Pf, Ps, Qs) :-
  suppress_min([[F, H, G], U], Pf, Pf2),
    suppress([U, [F, H, G] , UPere , A], Ps, Ps2),
    expand(U, LSuc, G),
    loop_successors(LSuc, Ps2, Pf2, Qs, PsF, PfF),
    insert([U, [F, H, G], UPere, A], Qs, Qs2),
    aetoile(PfF, PsF, Qs2).
```

Pour le prédicat aetoile on distingue 3 cas :

- Cas trivial 1 : Pf et Pu sont vides donc il n'y a aucune solution possible, on ne peut pas atteindre un état final
- Cas trivial 2 : On récupère F, le nœud min de Pf. Si F correspond à une situation finale alors on a trouvé une solution et on l'affiche.
- Cas général : D'abord on enlève F, le nœud min de Pf qui correspond à l'état U que l'on va développer et son équivalent dans Pu. Ensuite, on utilise expand pour récupérer la liste des successeurs de U et pour chacun on va calculer leur évaluation. Le prédicat loop_successors va traiter chaque successeur. Enfin, quand on a fini de développer U, qu'il est supprimé de P, on l'insère dans Q et on appelle la récursion avec les nouveaux arbres AVL.

c) Prédicats expand et loop_successors et affiche_solution

```
expand(U, LSuc, G) :-
    findall([S, [F, H, G2], U, Rule],
            (rule(Rule, C, U, S),
                   G2 is G+C,
                   heuristique(S, H),
                   F is G2+H),
            LSuc).
```

Le prédicat expand trouve tous les successeurs de l'état U et pour chaque il calcule l'heuristique et donc l'évaluation du coût sachant qu'on lui passe en paramètre G qui est le coût d'un coup.

```
loop_successors([], Ps,Pf,_,Ps,Pf).

loop_successors([[S, _, _, _]|LSuc], Ps, Pf, Qs, PsF, PfF) :-
    belongs([S, _, _, _], Qs),
    loop_successors(LSuc, Ps, Pf, Qs, PsF, PfF).


loop_successors([[S, [F, H, G], Pere, A]|LSuc], Ps, Pf, Qs, PsF, PfF) :-
    not(belongs([S, _, _, _], Qs)),
    belongs([S, [Fu, Hu, Gu], PereU, Au], Ps),
    F<Fu,
    suppress([S, [Fu, Hu, Gu], PereU, Au], Ps, Ps2),
    insert([S, [F, H, G], Pere, A], Ps2, Ps3),
    suppress([[Fu, Hu, Gu], S], Pf, Pf2),
    insert([[F, H, G], S], Pf2, Pf3),
    loop_successors(LSuc, Ps3, Pf3, Qs, PsF, PfF).

loop_successors([[S, [F, _, _], _, _]|LSuc], Ps, Pf, Qs, PsF, PfF) :-
    not(belongs([S, _, _, _], Qs)),
    belongs([S, [Fu, _, _], _, _], Ps),
    F>Fu,
    loop_successors(LSuc, Ps, Pf, Qs, PsF, PfF).

loop_successors([[S, [F, H, G], Pere, A]|LSuc], Ps, Pf, Qs, PsF, PfF) :-
    not(belongs([S, _, _, _], Qs)),
    not(belongs([S, _, _, _], Ps)),
    insert([S, [F, H, G], Pere, A], Ps, Ps2),
    insert([[F, H, G], S], Pf, Pf2),
    loop_successors(LSuc, Ps2, Pf2, Qs, PsF, PfF).
```

Le prédicat loop_successors couvre plusieurs cas. Le premier est la condition d'arrêt et correspond au cas où il n'y a plus de successeurs à explorer. Le deuxième vérifie si le successeur appartient à Q, si c'est le cas on ignore ce successeur car il a déjà été considéré. Le troisième cas vérifie si le successeur appartient à Ps. Si c'est le cas il compare l'évaluation de ce successeur passée en paramètre (F) et celle contenue dans Ps (Fu). Si F est inférieure cela signifie que le successeur considéré est meilleur donc on va l'ajouter dans Ps et Pf et supprimer l'ancien. En revanche si F est supérieur on ne fait rien on passe au successeur suivant car on a déjà une meilleure évaluation dans les arbres. Enfin, le dernier cas correspond au cas où le successeur n'appartient ni à Q ni à Ps. Dans ce cas, on l'insère dans Ps et Pf et on passe au suivant.

d) Analyse expérimentale

Nous avons utilisé le prédicat time pour connaître le temps d'exécution. Pour l'heuristique 1 nous n'obtenons des résultats que pour les 3 premières situations initiales et avec l'heuristique 2 nous avons des résultats pour les 4 premières.

Situation initiale	1	2	3	4	5
f*	2	5	10	20	30
H1	1	3	9	17	20
H2	1	3	5	837	906

Situations initiales :

1. [[a, b, c], [g, h, d], [vide, f, e]]

2. [[b, h, c], [a, f, d], [g, vide, e]]

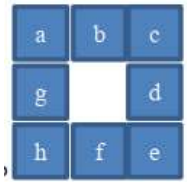
3. [[b, c, d], [a, vide, g], [f, h, e]]

4. [[f, g, a], [h, vide, b], [d, c, e]]

5. [[e, f, g], [d, vide, h], [c, b, a]]

Grâce à ces résultats, on peut observer que l'heuristique 2 (avec la distance de Manhattan) semble plus efficace car elle permet de résoudre une situation de plus que l'heuristique 1 et elle est plus rapide.

Le code n'est pas optimisé donc cela peut expliquer pourquoi nous n'avons pas des résultats pour toutes les situations.



Pour cette situation, aucune des deux heuristiques ne trouve de solution mais c'est explicable car cet état est non connexe avec l'état final. Il est donc impossible de trouver une solution.

TP2 Algo Min Max Application au Tic-Tac-Toe

Familiarisation avec le problème du tic-tac-toe 3x3

Quelle interprétation donnez-vous aux requêtes suivantes :

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

Une grille vide de tic-tac-toe (situation initiale) est stockée dans S et c'est le joueur J qui commence.

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

La situation initiale est dans S, on récupère la troisième ligne de S grâce au premier nth1 et grâce au deuxième nth1 il va modifier l'élément de la 2ème colonne de la 3ème ligne par un o.

Prédicat alignement :
Le prédicat alignement utilise les prédicats ligne, colonne et diagonale détaillés ci-contre :

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

colonne(C,M) :-
   colonne2(_, M, C).

colonne2(_N, [],[]).
colonne2(N, [L|R], [X|C]):-
    nth1(N, L, X),
    colonne2(N, R, C).

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).
```

Il permet de connaître tous les alignements gagnants possibles pour une matrice M donnée.
Dans le sujet on nous propose la matrice M = [[a,b,c],[d,e,f],[g,h,i]] et les résultats que nous obtenons (ci-dessous) sont cohérents et conformes au sujet.

Ali	
[a, b, c]	1
[d, e, f]	2
[g, h, i]	3
[a, d, g]	4
[b, e, h]	5
[c, f, i]	6
[a, e, i]	7
[c, e, g]	8

Solutions obtenues lors du test du prédicat alignement (Ali, M)

Ensuite, il faut déterminer si le joueur est capable de gagner avec l'alignement choisi. Pour cela on développe le prédicat possible qui vérifie si aucune case de l'alignement ne possède un symbole adverse ou s'il n'y a plus de cases libres. Le prédicat possible utilise le prédicat unifiable (qui vérifie qu'une case est encore libre), également développé ci-dessous. Pour ce TP, nous avons décidé de tous les deux coder dans deux fichiers distincts, le premier optimisé (Paul dans le GIT), le second non optimisé (Elise dans le GIT). Ceci dans le but d'étudier les différences de puissance entre les deux. Le principe d'optimisation étant simple, essayer au plus possible d'intégrer des cuts (ou des SI ALORS) au lieu de plusieurs clauses. De plus, nous avons mis en place des clauses de tests unitaires dans le fichier optimisé. Les tests étant relativement simples, ils ne seront pas détaillés dans ce rapport mais sont disponibles dans le GIT.

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

unifiable(X,_J) :-
    var(X).

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

Code des prédicats possible et unifiable (non optimisé)

```
unifiable(X,J) :-
  (var(X) ->
    true
  ;
    J == X
  ).
```

Code du prédicat unifiable (optimisé)

Enfin, à partir d'un alignement il faut déterminer s'il est potentiellement gagnant ou perdant. Pour cela on a développé les prédicats alignement_gagnant et alignement_perdant.

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

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

Code des prédicats alignement_gagnant et alignement_perdant

Alignement Ali	[x,x,x]	[x,o,x]	[o,o,o]
Résultat alignement_gagnant(Ali, x)	True	False	False
Résultat alignement_perdant(Ali, x)	False	False	True

Développement de l'heuristique h (Joueur, Situation)

L'heuristique permet d'évaluer les chances d'un joueur de gagner dans une situation donnée.
On calcule dans un premier temps AJ, les alignements potentiellement réalisables par J dans une situation donnée S. Ensuite on calcule AA, les alignements potentiellement réalisables par J dans S.

On a donc :

$$h = \begin{cases} 10000 \; si \; S \; est \; gagnante \; pour \; J \\ -10000 \; si \; S \; est \; perdante \; pour \; J \\ AJ - AA \; sinon \end{cases}$$

Voici le prédicat permettant de calculer l'heuristique :

```
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), !.

heuristique(J,Situation,H) :-
    adversaire(J, A),
    findall(D, (alignement(D, Situation), possible(D, J)), List),
    findall(D, (alignement(D, Situation), possible(D, A)), List2),
    length(List, NJ),
    length(List2, NA),
    H is NJ-NA.
```

Code du prédicat heuristique

NB : Dans le cas optimisé comme non optimisé les coupures sont présentes car elles sont nécessaires pour le bon fonctionnement de l'heuristique.

Pour tester le prédicat heuristique à partir de la situation initiale on peut utiliser la requête suivante :
`?- situation_initiale(S), joueur_initial(J), heuristique (J, S, H).`
A la suite de cette requête on obtient H = 0 car le joueur a autant de chances de gagner que son adversaire donc AA et AJ s'annulent.

Requête avec une situation où J est gagnant :
`?- S= [[x,o,x],[o,x,o],[x,o,x]], joueur_initial(J), heuristique (J, S, H).`
Ici, on s'attend à ce que H = 10 000 car c'est une situation gagnante.
Résultat obtenu : H = 10000

Requête avec une situation où J est perdant :
`?- S= [[x,o,x],[o,o,x],[x,o,o]], joueur_initial(J), heuristique (J, S, H).`
Ici, on s'attend à ce que H = -10 000 car c'est une situation perdante.
Résultat obtenu : H = -10000

Requête avec une situation nulle :
`?- S= [[x,x,o],[o,o,x],[x,o,o]], joueur_initial(J), heuristique (J, S, H).`
Ici, on s'attend à ce que H = 0 car c'est une situation nulle qui n'avantage personne.
Résultat obtenu : H = 0

Développement de l'algorithme Négamax

Pour l'algorithme de négamax il y a 3 situations possibles :

1. La profondeur maximale est atteinte : on ne peut plus développer l'état, donc on ne peut plus jouer et on évalue l'état avec l'heuristique
2. La profondeur maximale n'est pas atteinte mais le joueur ne peut pas jouer (ici, pour le jeu du Tic-Tac-Toe c'est quand la grille est complète). Il n'y a pas de coup à jouer et on évalue l'état avec l'heuristique.
3. La profondeur maximale n'est pas atteinte et le joueur peut encore jouer. Il faut chercher d'abord tous les coups possibles associés à des situations qui suivront (prédicat successeurs). Ensuite pour chaque successeur, on applique négamax sur la situation associée (loop_negamax) et on obtient une liste de couples de coups et de valeurs.

```
loop_negamax(_,_, _  ,[],              []).
loop_negamax(J,P,Pmax,[[Coup,Suiv]|Succ],[[Coup,Vsuiv]|Reste_Couples]) :-
  loop_negamax(J,P,Pmax,Succ,Reste_Couples),
  adversaire(J,A),
  Pnew is P+1,
  negamax(A,Suiv,Pnew,Pmax, [_,Vsuiv]).
```

Enfin, parmi cette liste on garde le meilleur couple (prédicat meilleur (version optimisée à gauche)) c'est-à-dire celui qui a la plus petite valeur.

```
meilleur([A], A).
meilleur([[CX, VX]|T],MeilleurCouple) :-
    T \= [],
    meilleur(T, [_, VY]),
    VX=<VY,
    MeilleurCouple = [CX, VX].


meilleur([[_, VX]|T], MeilleurCouple) :-
    T \= [],
    meilleur(T, [CY, VY]),
    VX>VY,
    MeilleurCouple = [CY, VY].
```

```
meilleur([X], X).
meilleur([[_, V] | Rest], [C1, V1]) :-
  Rest \= [],
  meilleur(Rest, [C1, V1]),
  V >= V1,
  !.
meilleur([[C, V] | Rest], [C, V]) :-
  Rest \= [].
```

L'algorithme va retourner le couple associant le coup obtenu et -V (V étant la valeur du coup).

```
/*Cas 1 : on est à la profondeur max*/
negamax(J, Etat, Pmax, Pmax, [nil, Val]) :-
    heuristique(J, Etat, Val).

/*Cas 2 : J ne peut pas jouer (tableau complet)*/
negamax(J, Etat, P, Pmax, [nil, Val]) :-
    P \= Pmax,
    situation_terminale(J, Etat),
    heuristique(J, Etat, Val).

/*Cas 3 : Profondeur max pas atteinte J peut encore jouer*/
negamax(J, Etat, P, Pmax, [Coup, Val]) :-
    P\=Pmax,
    not(situation_terminale(J, Etat)),
    successeurs(J, Etat, Succ),
    loop_negamax(J, P, Pmax, Succ, ListeCouples),
    meilleur(ListeCouples, [Coup, V1]),
    Val is -V1.
```

Code du prédicat negamax (non optimisé)

```
negamax(J, Etat, P, P, [nil, H]) :-
  heuristique(J,Etat,H), !.

negamax(J, Etat, _, _, [nil, H]) :-
  situation_terminale(J, Etat),
  heuristique(J,Etat,H), !.

negamax(J, Etat, _, _, [nil, H]) :-
  heuristique(J,Etat,H),
  H = 10000, !.

negamax(J, Etat, _, _, [nil, H]) :-
  heuristique(J,Etat,H),
  H = -10000, !.

negamax(J, Etat, P, Pmax, [Coup, V2]) :-
  P < Pmax,
  successeurs(J, Etat, Succ),
  loop_negamax(J, P, Pmax, Succ, L),
  meilleur(L, [Coup, V1]),
  V2 is 0 - V1.
```

Code du prédicat negamax (optimisé)

Expérimentation et extensions

Tableau représentant les tests avec une situation initiale vide et un joueur x (code non optimisé).

Profondeur	1	2	3	4	5	6	7	8	9	10
B	nil	[2,2]	[2,2]	[2,2]	[2,2]	[2,2]	[2,2]	[2,2]	[3,3]	[3,3]
V	0	4	1	3	1	3	1	2	0	0
Temps	0s	0,003s	0,016s	0,076s	0,453s	3,245s	18,6	Out of local stack		
Temps (Opti)	0s	0,001s	0,013s	0,046s	0,259s	1,264s	5,23	15,8	32,0	47,8

Lorsque la profondeur est égale à 1, on ne cherche pas le coup suivant donc on obtient nul comme coup à jouer puisque l'on n'a rien exploré. Ensuite, comme la matrice est vide au départ dans la situation initiale, il est logique d'obtenir comme meilleur coup la place centrale car c'est celle qui optimise le nombre possible de lignes à faire.

Cependant, à partir d'une profondeur 9 (c'est-à-dire 8 coups de joués), le coup préconisé change. Cela vient du fait qu'au morpion, on ne peut jamais perdre (si l'on joue bien). Négamax ne le voyait pas au début car il n'allait pas assez profond dans l'arbre des possibilités, donc, il proposait l'état [2,2] qui est a priori le meilleur. Dans les deux dernières cases du tableau, négamax se rend compte qu'en réalité il y aura toujours égalité (il suppose que l'adversaire est un cador). Donc il préconise un coup au hasard car tous les coups sont équivalents, l'implémentation de l'algorithme fait que c'est le dernier de la liste des successeurs (donc [3,3]).

En ce qui concerne l'optimisation, nous voyons nettement la différence à partir de la profondeur 6, et d'autant plus à partir de la profondeur 8 où l'algo ne peut même pas terminer. Cela prouve bel et bien l'importance des cuts bien placés qui élaguent l'arbre de dérivation, réduisant sa taille, et donc son espace de stockage et sa durée de création.

Voici un test dans une autre situation que la matrice vide :

Situation : X O X
X X O
_ _ O

```
main(B,V, Pmax) :-
    S= [[x,o,x], [x,x,o],[_,_,o]],
    joueur_initial(J),
    negamax(J, S, 1, Pmax, [B, V]).
```

Code du main pour tester cette situation

B = [3, 1],
V = 10000

Retour de l'algorithme

On voit ici que l'algorithme conseille de jouer en [3,1] ce qui est une très bonne situation puisqu'elle permet au joueur x de gagner (d'où la valeur 10 000 associée au coup).

Enfin, en conclusion à ce TP nous voudrions souligner la différence d'efficacité entre les algorithmes optimisés et non optimisés. En effet, les optimisations apportées ont permis d'obtenir de meilleures performances. Cependant ceci aurait pu être encore plus optimisé : par exemple, prendre en compte les symétries et ainsi éviter de traiter plusieurs fois les mêmes cas. Ces optimisations auraient pu améliorer les performances en termes de temps d'exécution mais aussi en termes d'utilisation de la pile. Le code optimisé fonctionne avec une profondeur de 9, mais la prise en compte des symétries baisserait le temps de réponse (actuellement de quelques secondes et donc visible par l'utilisateur). Cependant, une profondeur très élevée n'est pas forcément la meilleure façon d'obtenir un algorithme performant. Affiner les heuristiques est aussi un moyen de prédire le meilleur coup à jouer. Par exemple, dans le cadre du Tic-Tac-Toe, on aurait pu allouer un certain score lorsque le coup permet d'aligner deux symboles identiques.

Pour aller plus loin : Jouons au morpion !

Nous avons trouvé dommage d'avoir développé négamax qui permet d'obtenir le meilleur coup sans pour autant l'utiliser. Nous avons donc décider de coder un petit interpréteur qui permet de jouer contre notre algorithme. Cet interpréteur est disponible dans le répertoire Paul du GIT, dans le fichier morpion.pl. Il s'agit essentiellement de prédicats dédiés à l'affichage et à la lecture du coup de l'utilisateur. Nous avons ensuite défini deux prédicats, un pour jouer en local et un pour jouer avec négamax. Et enfin le dernier prédicat (**play(Profondeur)**) qui fait alterner les deux.

On peut régler la profondeur maximale de négamax avec Profondeur, et donc régler la difficulté de la partie.

Pour jouer, il suffit de charger le fichier et de lancer play. Pour saisir un coup, il faut taper le numéro de la ligne (entre 1 et 3), un point (.) et appuyer sur Enter pour valider. Il faut procéder de même pour la colonne (1-3, puis ., puis Enter).

Si vous le souhaitez, vous pouvez récupérer ce fichier et le rajouter sur Moodle, le code n'étant pas très intéressant à développer pour les étudiants. Cependant, cela leur permettrait d'utiliser leur algorithme négamax dans des conditions amusantes.