COMPTE RENDU TP1 IA

Partie 1 : Familiarisation avec le problème du Taquin 3×3

1.2.a. La clause Prolog permettant de représenter la situation finale du Taquin 4x4 est :

```
final_state_4([[1, 2, 3, 4],
               [5, 6, 7, 8],
               [9, 10, 11, 12],
               [13, 14, 15, vide]]).
```

1.2.b. A quelles questions permettent de répondre les requêtes suivantes :

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

Cette requête permet de répondre à la question "Quelles sont les coordonnées de la pièce d dans la situation initiale ?". En effet, elle renvoie la ligne et la colonne de la pièce d dans la matrice de situation initiale nommée ici Ini.

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

Cette ligne fait le contraire : on cherche quelle pièce se trouve à la ligne 3 et colonne 2 de la matrice Fin représentant la situation finale.

1.2.c. La requête Prolog permettant de savoir si une pièce donnée P est bien placée dans U_0 (par rapport à Fin) est :

```
initial_state(Ini), final_state(Fin), nth1(L, Ini, Ligne), nth1(C, Ligne, P),
nth1(L, Fin, Ligne_fin), nth1(C, Ligne_fin, P).
```

1.2.d. On peut utiliser les 3 requêtes suivantes pour trouver une situation suivante de l'état initial du Taquin 3x3 :

```
initial_state(Ini), rule(up, 1, Ini, Suiv).
initial_state(Ini), rule(right, 1, Ini, Suiv).
initial_state(Ini), rule(left, 1, Ini, Suiv).
```

On ne peut pas utiliser "down" car le vide est déjà en bas du taquin pour la situation initiale.

1.2.e. La requête qui permet d'avoir ces 3 réponses regroupées dans une liste est :

```
initial_state(Ini), findall(Suiv, rule(X, 1, Ini, Suiv), L).
```

1.2.f. La requête qui permet d'avoir la liste de tous les couples $[A,S]$ tels que S est la situation qui résulte de l'action A en U_0 est :

```
initial_state(Ini), findall([X, Suiv], rule(X, 1, Ini, Suiv), L).
```

Partie 2 : Développement des 2 heuristiques

2.1 Voici le code correspondant à l'heuristique du nombre de pièces mal placées.

```
heuristique1(U, H) :-
    findall(X, (piece_placee(U, X),X\=vide), Liste_placees),
    length(Liste_placees, Nb),
    H is (8 - Nb).
```

```
piece_placee(U, P) :-
    final_state(Fin),
    nth1(L, U, Ligne),
    nth1(C, Ligne, P),
    nth1(L, Fin, Ligne_fin),
    nth1(C, Ligne_fin, P).
```

2.2 Voici le code correspondant à l'heuristique basée sur la distance de Manhattan

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

heuristique2(U, H) :-
      final_state(Fin),
      findall(Elt,(coordonnees([L,C], U, Elt), coordonnees([Lf,Cf], Fin, Elt),
Elt \= vide, [L, C] \= [Lf, Cf]), List_mal_placees),
      findall(Dist, (member(X,List_mal_placees), distance_man(U, Fin, X,
Dist)), List_dist),
      sumlist(List_dist, H).

distance_man(U, F, Elt, Dist) :-
        coordonnees([L,C], U, Elt),
        coordonnees([Lf,Cf], F, Elt),
        Dist is (abs(Lf - L) + abs(Cf - C)).
```

Partie : implémentation de A*

3.2 Algorithme A* adapté aux structures AVL choisies

Voici le code correspondant au prédicat `affiche_solution` :

```
%cas trivial
affiche_solution([U,_Vals,_, _], _Q):-
    print("AFFICHE_SOLUTION - CAS TRIVIAL "), nl,
    initial_state(U),    % on est revenu dans l'etat initial
    write_state(U),nl,!.

affiche_solution([U,_Vals,P, Apu ],Q):-
    print("AFFICHE_SOLUTION - CAS GENERAL "),
    nl,
    belongs([P,ValsP,GP,AP], Q),
    affiche_solution([P,ValsP,GP,AP], Q),
    write("Action : "),
    write(Apu), nl, nl,
    write_state(U),nl.
```

Voici le code correspondant au prédicat `expand` :

```
%Determiner tous les etats ayant U pour pere
%et Calculer leurs évaluations
expand(Upere,G,List_suiv):-
```

```
    %print("EXPAND"),nl,
    findall([USuiv,[Fs,Hs,Gs],X], (rule(X,Cout,Upere,USuiv),Gs is
G+Cout, heuristique(USuiv,Hs), Fs is Hs+Gs), List_suiv).
```

Voici le code correspondant au prédicat `loop_successors` :

```
%Traiter chaque noeud successeur
%% Cas trivial : liste vide
loop_successors([], _, _,Pf, Pu, Pf, Pu) :-
    %write("LOOP_SUCCESSORS CAS VIDE"), nl,
    !.

%Cas 1 : Si S est connu dans Q alors oublier cet etat
loop_successors([[S,_,_] | Tail], U, Q, Pf, Pu, Pf_new, Pu_new) :-
    %print("PROCESS_SUCCESSOR - CAS 1"), nl,
    belongs([S,_,_,_], Q),
    loop_successors(Tail, U, Q, Pf, Pu,Pf_new, Pu_new), !.

%Cas 2 : Si S est connu dans Pu et que S est mieux que l'ancienne
valeurs dans l'arbre
loop_successors([[S,[Fs, Hs, Gs],A]|Tail], U, Q, Pf, Pu, Pf_new,
Pu_new) :-
    %print("PROCESS_SUCCESSOR - CAS 2"), nl,
    belongs([S,[F, _, _],_,_],Pu),
    F > Fs,
    suppress([S,_,_,_], Pu, Pu_aux),
    insert([S,[Fs,Hs,Gs],U,A], Pu_aux, Pu_aux2),
    suppress([S,_,_,_], Pf, Pf_aux),
    insert([[Fs,Hs,Gs],S], Pf_aux, Pf_aux2),
    loop_successors(Tail, U, Q, Pf_aux2, Pu_aux2, Pf_new, Pu_new),!.

%Cas 3 : Si S est connu dans Pu et que S n'est pas mieux que l'ancienne
valeurs dans l'arbre
loop_successors([[S,[Fs, _, _],_]|Tail], U,  Q,Pf, Pu, Pf_new, Pu_new)
:-
    %print("PROCESS_SUCCESSOR - CAS 3"), nl,
    belongs([S,[F, _, _],_,_],Pu),
    F =< Fs,
    %On ne change pas le noeud dans pf et pu car pas mieux que le
precedent
    loop_successors(Tail, U, Q, Pf, Pu, Pf_new, Pu_new),!.

%Cas 4 : Si S est une situation nouvelle
loop_successors([[S,[Fs, Hs, Gs],A]|Tail], U,  Q,Pf, Pu,Pf_new, Pu_new)
:-
    %print("PROCESS_SUCCESSOR - CAS 4"), nl,
    insert([S,[Fs, Hs, Gs],U,A], Pu, Pu_aux),
    insert([[Fs, Hs, Gs],S], Pf, Pf_aux),
    loop_successors(Tail, U, Q, Pf_aux, Pu_aux, Pf_new, Pu_new).
```

3.3 Analyse expérimentale

Nous avons utilisé le prédicat **`time/1`** pour calculer le temps d'exécution de A*. Pour la situation initiale 5, nous ne trouvons de solution ni avec h1 ni avec h2 sans dépasser la taille maximale de la pile.

Situation initiale	Temps d'exécution avec h1 en ms	Temps d'exécution avec h2 en ms
Situation initiale 1	5	3
Situation initiale 2	3	2
Situation initiale 3	9	4
Situation initiale 4	1086	494
Situation initiale 5	Stack limit	Stack limit

Pour la situation initiale non connexe, l'algorithme échoue sans trouver de solution après 704 ms avec h1 et 838 ms avec h2.

Pour représenter un Rubik's Cube, nous pourrions faire une liste de "6 taquins" représentant chacune des faces (une lettre par couleur).
Exemple : la situation finale ressemblerait à :
[[[r,r,r],[r,r,r],[r,r,r]], [[b,b,b],[b,b,b],[b,b,b]],...]
On modifie donc le prédicat `successeurs` en conséquence.

COMPTE RENDU TP2 IA

Partie 1 : Familiarisation avec le problème du TicTacToe 3×3

1.2 Quelle interprétation donnez-vous aux requêtes suivantes :

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

On définit la situation initiale : il s'agit de la grille vide. On récupère le symbole utilisé par le joueur qui commence : c'est x.

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

On définit la situation initiale (grille vide) puis on place un 'o' à la troisième ligne deuxième colonne.

2.2 Nous définissons les différentes formes d'alignements suivantes :

Les lignes :

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

La ligne i correspond à l'élément i de la matrice, qui est de cette forme : $matrice = [ligne_1, ligne_2, \dots, ligne_n]$

Les colonnes :

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

colonne_aux([], [], _).

colonne_aux([HC|RC], [HM|RM], N) :-
    nth1(N, HM, HC),
    colonne_aux(RC, RM, N).
```

On parcourt la matrice ligne par ligne et on stocke le $i^{ème}$ élément de chacune des lignes. Une fois arrivé à la dernière ligne, nous avons obtenu la colonne i et on passe à la colonne suivante, la $(i+1)^{ème}$.

Les deux diagonales :

```
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).
```

La première diagonale était déjà codée. Pour la seconde, nous avons fait pareil que la première mais on décrémente K pour parcourir la matrice dans l'autre sens.

Ainsi, nous obtenu bien les 8 alignements possibles d'une matrice 3x3.

```
?- M = [[a,b,c],[d,e,f],[g,h,i]], alignement(Ali, M).
M = [[a, b, c], [d, e, f], [g, h, i]],
Ali = [a, b, c] ;
M = [[a, b, c], [d, e, f], [g, h, i]],
Ali = [d, e, f] ;
M = [[a, b, c], [d, e, f], [g, h, i]],
Ali = [g, h, i] ;
M = [[a, b, c], [d, e, f], [g, h, i]],
Ali = [a, d, g] ;
M = [[a, b, c], [d, e, f], [g, h, i]],
Ali = [b, e, h] ;
M = [[a, b, c], [d, e, f], [g, h, i]],
Ali = [c, f, i] ;
M = [[a, b, c], [d, e, f], [g, h, i]],
Ali = [a, e, i] ;
M = [[a, b, c], [d, e, f], [g, h, i]],
Ali = [c, e, g] ;
false.
```

Pour savoir si un alignement est encore possible, nous devons savoir si X est unifiable à J.

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

possible([],_).
```

X est unifiable à J si X n'est pas libre et que sa valeur correspond à celle de J OU si X est libre. Nous utilisons les prédicats déjà définis en Prolog `var(X)` et `ground(X)`.

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

Un alignement est gagnant pour J si toutes les cases ont une valeur et que cette valeur est celle du joueur J. On peut aussi se dire qu'un alignement est gagnant s'il est possible et que toutes les cases ont une valeur.

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

Un alignement est perdant pour J s'il est gagnant pour son adversaire :

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

Voir le fichier `tictactoe.pl` pour les tests unitaires.

Partie 2 : Développement de l'heuristique h(Joueur, Situation)

Les cas où un des joueurs est gagnant ont déjà été définis. On définit le dernier cas : on cherche tous les alignements possibles pour le joueur J, puis pour son adversaire et on les stocke dans deux listes différentes. Ensuite, on soustrait la taille de la liste des alignements possibles pour l'adversaire à la taille de la liste des alignements possibles pour le joueur J.

```
heuristique(J,Situation,H) :-
    findall(AligJ,(alignement(AligJ,Situation),possible(AligJ,J)),ListePos),
    adversaire(J,I),
    findall(AligI,(alignement(AligI,Situation),
    possible(AligI,I)),ListePosAdv),
    length(ListePos,L1),
    length(ListePosAdv,L2),
    H is L1 - L2.
```

Voir le fichier `tictactoe.pl` pour les tests unitaires.

Partie 3 : Développement de l'algorithme Negamax

3.2 Le prédicat qui permet de connaître sous forme de liste l'ensemble des couples [Coord, Situation_Resultante] tels que chaque élément (couple) associe le coup d'un joueur et la situation qui en résulte à partir d'une situation donnée est `successeurs/3`.

3.3 Voir le fichier `negamax.pl`.

Partie 4 : Expérimentation et extensions

4.1

PROFONDEUR	B	V
1	[]	0
2	B = [2, 2]	4
3	B = [2, 2]	1
4	B = [2, 2]	3
5	B = [2, 2]	1
6	B = [2, 2]	3
7	B = [2, 2]	1
8	B = [2, 2]	2
9	B = [3, 3]	10000

Figure : résultats obtenus avec Negamax en fonction de la profondeur

Tant que la profondeur n'est pas suffisante pour explorer toutes les possibilités, le meilleur coup à jouer est [2, 2]. Cependant, avec la profondeur 9, le joueur gagne et le meilleur coup est finalement [3, 3]. Avec la profondeur 9, toute la grille est remplie

et on a donc fini la partie. On a donc le recul nécessaire pour connaître le meilleur coup à jouer pour être sûr de gagner.

4.2 Pour éviter de développer inutilement des situations, il faudrait stocker les situations déjà développées précédemment. Ainsi, on peut vérifier si le successeur et ses symétriques ont déjà été explorés. Pour les symétriques, il suffit de faire 3 rotations : 90°, 180° et 270°, ainsi qu'une symétrie en miroir : on lit chaque ligne de gauche à droite et on les réécrit de droite à gauche.

4.3 Pour le puissance 4, il faudrait agrandir la grille de jeu. Ensuite, on redéfinirait les alignements pour qu'ils ne correspondent plus forcément à une ligne/colonne/diagonale entière mais seulement à 4 jetons.
Il faudrait aussi revoir le prédicat `successeurs/3` pour ne pas pouvoir mettre de jetons en haut de la grille si les cases situées dessous ne sont pas remplies.
Le reste n'a pas besoin d'être modifié.

4.4 Pour éviter d'explorer des branches inutilement, nous pouvons appliquer l'algorithme Alpha-Beta. Pour cela, on utilise une borne minimale α pour les nœuds Max et une borne maximale β pour les nœuds Min. On transmet la valeur de la borne au nœud fils. Lorsqu'on arrive à une feuille, on a : $\alpha = \beta = h(u)$.
On élague une branche quand un fils remonte une valeur α $(resp. \beta)$ plus grande que la valeur β $(resp. \alpha)$ de son père.