Valentin GUERBETTE, 4IR B2

Compte-Rendu de TP d'IA

I) TP1 : A*

1.2) a) Pour représenter la situation finale du Taquin 4x4 :

`?- final_state(Fin), nth1(4,Fin,Ligne), nth1(4,Ligne,P), P = vide.`

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

Cette requête permet de donner l'état d'une cellule de l'état initial.

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

Cette requête permet de vérifier la 2eme cellule de la 3eme ligne de l'état final.

c) Pour savoir si une pièce donnée P (ex : a) est bien placée dans U0 :

`?- initial_state(Ini), nth1(3,Ini,Ligne), nth1(2,Ligne,P), P = vide.`

d) Pour trouver une situation suivante de l'état initial du Taquin 3×3 :

`?- initial_state(Ini), rule(up, 1, Ini, Next).`

`?- initial_state(Ini), rule(right, 1, Ini, Next).`

`?- initial_state(Ini), rule(left, 1, Ini, Next).`

e) Pour toutes les regrouper dans une liste :

```
?- initial_state(Ini), findall(Next, rule(Move, 1, Ini, Next),
Next_moves) .
```

f) Pour 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([Next, Move], rule(Move, 1, Ini, Next),
Next_moves) .
```

2) Développement de 2 heuristiques:

La première heuristique que nous utiliserons est définie par le nombre de pièces mal placées relativement à la situation finale.

heuristique1 est bien coïncidente, càd nulle quand évaluée pour l'état final :

```
| ?- final_state(Final), heuristique1(Final,H).

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

On teste de même l'autre cas extrême:

```
| ?- initial_state(Ini), heuristique1(Ini,H).

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

Nous passons désormais à la seconde heuristique qui repose sur la somme des distances de Manhattan séparant chaque pièce de son emplacement dans la situation finale.

On vérifie bien qu'elle soit coïncidente :

```
| ?- final_state(Final), heuristique2(Final,H).

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

On obtient une valeur de 6 depuis l'état initial :

```
| ?- initial_state(Ini), heuristique2(Ini,H).

H = 6
Ini = [[b,h,c],[a,f,d],[g,vide,e]]
```

II) TP2 : Negamax

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

La partie débute.

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

Le joueur o se place aux coordonnées (3,2) en début de partie.

Pour accéder à une ligne d'une matrice carrée NxN :

Il suffit d'itérer dans la liste qui compose la matrice. En l'occurrence, on vérifie juste son appartenance :

```
ligne(L, M) :- member(L,M).
```

Pour accéder à une colonne d'une matrice carrée NxN :

On peut passer par une fonction auxiliaire `get_colonne` afin de conserver l'index de la colonne au cours des itérations :

```
get_colonne(_,[],[]).

get_colonne(Index,[HM|TM],[HC|TC]) :-

    nth1(Index, HM, HC),

    get_colonne(Index, TM, TC).

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

Pour accéder aux diagonales d'une matrice carrée NxN :

Il n'y a que deux diagonales dans un matrice carrée :

La première va du coin en haut à gauche de la matrice jusqu'au coin en bas à droite. Pour l'obtenir, il faut incrémenter l'indice de 1 à N :

```
premiere_diag(_,[],[]).
```

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

La seconde diagonale va du coin en haut à droit jusqu'au coin en bas à gauche, il faut donc décrémenter l'indice au cours des appels récursifs :

```
seconde_diag(0,[],[]).
seconde_diag(K,[E|D],[Ligne|M]) :-
      nth1(K,Ligne,E),
      K0 is K-1,
      seconde_diag(K0,D,M).
```

Au final :

```
diagonale(D, M) :-
      premiere_diag(1,D,M).
diagonale(D, [HM|TL]) :-
      length(HM,N),
      seconde_diag(N,D,[HM|TL]).
```

On utilise ces prédicats pour identifier tous les alignements possible sur une matrice :

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

Ali = [a,b,c]
M = [[a,b,c],[d,e,f],[g,h,i]] ? a

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]
M = [[a,b,c],[d,e,f],[g,h,i]]

no
```

Par la suite, nous définissons les alignements gagnants et perdants. Ceux-ci vérifient les tests suivants :

```
| ?- alignement_gagnant([x,_,x],x).

yes


| ?- alignement_perdant([x,_,x],o).

yes
```

2) Développement de l'heuristique

Notre heuristique est définie comme la différence entre le nombre de coup possible pour le joueur courant et l'adversaire. Dans le cas d'une victoire, elle est également à +10000 approximant +inf et dans le cas d'une défaite à -10000 approximant –inf.

Ces valeurs seront essentielles au déroulement de l'algorithme minmax avec convention negamax.

```
nb_ali(J,Situation,N) :-

      findall(Ali, ( alignement(Ali,Situation), possible(Ali,J) ) ,
Alignements),

      length(Alignements,N).


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) :-
    nb_ali(J,Situation,NbAliJ),
    adversaire(J,A),
    nb_ali(A,Situation,NbAliA),
    H is NbAliJ-NbAliA, !.
```