INSA TOULOUSE

Rapport TP Intelligence Artificielle

UF : Systèmes Intelligents

Brandon ZHONG 4IR B1

Partie 1 : Algorithme A*- Taquin

1) Familiarisation avec le problème du Taquin 3×3

1.2.a) initial_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).

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

L'élement d dans la colonne C de la ligne L.
Vérifie le P de la colonne 2 dans la ligne 3.

c) initial_state(U0), coordonnees([L,C], U0, a),

final_state(F), coordonnees([L,C], F, a)).

d) initial_state(U0), rule(X,1,U0,U1). (On a trois rules seulement car on ne peut plus faire « down » depuis cette situation.)

```
?- initial_state(U0), rule(X, 1, U0, U1).
U0 = [[b, h, c], [a, f, d], [g, vide, e]]
X = up
U1 = [[b, h, c], [a, vide, d], [g, f, e]]
Yes (0.00s cpu, solution 1, maybe more)
U0 = [[b, h, c], [a, f, d], [g, vide, e]]
X = left
U1 = [[b, h, c], [a, f, d], [vide, g, e]]
Yes (0.00s cpu, solution 2, maybe more)
U0 = [[b, h, c], [a, f, d], [g, vide, e]]
X = right
U1 = [[b, h, c], [a, f, d], [g, e, vide]]
Yes (0.00s cpu, solution 3, maybe more)
No (0.00s cpu)
```

e) initial_state(U0), findall(X, rule(X,1,U0,U1),L).

f) initial_state(U0), findall([A,S], rule(A,1,U0,S),L).

```
?- initial_state(U0), findall([A, S], rule(A, 1, U0, S), L).
U0 = [[b, h, c], [a, f, d], [g, vide, e]]
A = A
S = S
L = [[up, [[b, h, c], [a, vide, d], [g, f, e]]], [left, [[b, h, c], [a, f, d], [vide, g, e]]], [right, [[b, h, c], [a, f, d], [g, e, vide]]]]
Yes (0.00s cpu)
```

2) Développement des 2 heuristiques

2.1.a) initial_state(U0),final_state(F),
findall(Elt, (mal_place(Elt,U0,F),Elt\=vide), L).
taille : initial_state(U0),final_state(F),
findall(Elt, (mal_place(Elt,U0,F),Elt\=vide), L), length(L,T).

2.2)

Aller à la fin du fichier `taquin.pl` et coder les prédicats logiques `heuristique1(U,H)` et `heuristique2(U,H)`.
Ces prédicats ont été temporairement implémentés par un "bouchon" (`true`) pour passer la compilation.

Tester les deux heuristiques sur les deux situations extrêmes (U_0 et F).

initial_state(U0), dm(c,U0,D).
initial_state(U0), heuristique1(U0, H). H = 4
initial_state(U0), heuristique2(U0, H). H = 5

```
?- final_state(F), heuristique1(F, H).
No (0.00s cpu)
?- final_state(F), heuristique2(F, H).
F = [[a, b, c], [h, vide, d], [g, f, e]]
H = 0
Yes (0.00s cpu)
```

3) Implémentation de A*

3.2) Situation S0: avec F0=H0+G0.

Tout est précisé dans le fichier aetoile avec des tests unitaires pour quelques prédicats. Pour tester A*, il suffit de taper « main. » puis more pour avoir toutes les réponses possibles (en fonction des situations initiales).
Heuristique2 a une meilleure performance après comparaison des résultats.

Partie 2 : Négamax – TicTacToe

1) Familiarisation avec le problème du TicTacToe 3×3

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

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

1.2) -Trouver le joueur initial pout la situation.
-Place à la ligne 3 colonne 2 le o.

2.2) J'ai complété les différentes alignements possibles dans le fichier tictactoe.pl pour retrouver les 8 alignements après la requête.

```
?- M = [[a,b,c], [d,e,f], [g,h,i]],   alignement(Ali, M).
Ali=[a,b,c];
Ali=[d,e,f];
Ali=[g,h,i];
Ali=[a,d,g];
Ali=[b,e,h];
Ali=[c,f,i];
Ali=[a,e,i];
Ali=[c,e,g];
no
```

2.2) Quelques tests unitaire pour répondre aux différents cas :

Définir enfin les prédicats **`alignement_gagnant(A, J)`** et **`alignement_perdant(A, J)`** qui réussissent si A est un alignement totalement instancié (utiliser le prédicat **`ground`** pour le savoir) ne contenant que des valeurs **`J`** (respectivement que des valeurs de l'adversaire de **`J`**).

Proposer des requêtes de tests unitaires pour chaque prédicat.

A=[o,o,o], alignement_perdant(A,x)	true
A=[o,o,o], alignement_perdant(A,o)	false
A=[o,o,o], alignement_gagnant(A,o)	true
A=[o,o,o], alignement_gagnant(A,x)	false
A=[o,_,o], alignement_gagnant(A,o)	false

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

2) Petite vérification de la valeur de h après les implémentations.

situation_initiale(S), heuristique(J,S,H). h=0

3) Développement de l'algorithme Negamax

```
?- main(B, V, 0).
B = rien
V = 0
Yes (0.00s cpu)
?- main(B, V, 1).
B = [2, 2]
V = 4
Yes (0.00s cpu, solution 1, maybe more)
?- main(B, V, 2).
B = [2, 2]
V = 1
Yes (0.00s cpu, solution 1, maybe more)
?- main(B, V, 3).
B = [2, 2]
V = 3
Yes (0.02s cpu, solution 1, maybe more)
```

```
?- main(B, V, 4).
B = [2, 2]
V = 1
Yes (0.05s cpu, solution 1, maybe more)
?- main(B, V, 5).
B = [2, 2]
V = 3
Yes (0.25s cpu, solution 1, maybe more)
No (0.25s cpu)
?- main(B, V, 6).
B = [2, 2]
V = 1
Yes (0.90s cpu, solution 1, maybe more)
No (0.92s cpu)
```

```
?- main(B, V, 7).
B = [2, 2]
V = 2
Yes (2.32s cpu, solution 1, maybe more)
?- main(B, V, 8).
Abort
```

Après avoir implémenté les différentes, nous procédons au test pour vérifier le bon fonctionnement et le temps d'éxécution qui s'amplifie au fur et à mesure que le jeu avance (1..9). De plus, on voit bien que avec cette implémentation, mon code ne permet pas d'arriver jusqu'à la fin du jeu (8 et 9), car le temps d'éxécution est trop grande.

Sur les deux TP, il est possible d'améliorer la performance des prédicats. Cependant, le temps d'exécution est correct et tous les prédicats fonctionnent correctement.