# Rapport IA – TP1

|  |
| --- |
| Table des matières  [Rapport IA – TP1 1](#_Toc65232086)  [Familiarisation avec le problème du Taquin 3\*3 1](#_Toc65232087)  [Developpement des deux heuristiques 2](#_Toc65232088)  [Fonction utiliser pour calculer le temps CPU 2](#_Toc65232089)  [Heuristique 1 2](#_Toc65232090)  [Heuristique 2 2](#_Toc65232091)  [Extensions prévues ou entrevues 2](#_Toc65232092)  [Rapport IA – TP2 2](#_Toc65232093) Familiarisation avec le problème du Taquin 3\*3 |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |
|  |

a) La cause prolog permettant de représenter la situation finale du taquin 4\*4 est :

final\_state

([[1, 2, 3, 4],

[5,6,7,8],

[9,10,11,12],

[13,14,15,vide]).

b) ?- initial\_state(Ini), nth1(L,Ini,Ligne), nth1(C,Ligne, d).

La première requête permet de connaitre la place de d dans l'état initial. C étant la colonne et L la ligne.

?- final\_state(Fin), nth1(3,Fin,Ligne), nth1(2,Ligne,P)

La deuxième requête permet de connaitre la lettre qui se trouve à la ligne 2 et colonne 3 dans l'état final.

c) Cette requête vérifie si a est à la même place dans l'état initial et dans l'état final:

?- initial\_state(Ini), nth1(L, Ini, Ligne), nth1(C, Ligne, a), final\_state(Fin), nth1(L, Fin, Ligne2), nth1(C, Ligne2, a).

d) ?- initial\_state(Ini), rule(\_,1,Ini,Next).

Cette requête permet de trouver une situation suivante à l’état initiale du taquin.

e) ?- initial\_state(Ini), findall(X, rule(X,1,Ini,Next),Y).

Cette requête permet d’avoir les réponses précédentes dans une liste.

f) ?- initial\_state(Ini), findall([X,Next], rule(X,1,Ini,Next),Y).

Cette 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 Uo.

## Developpement des deux heuristiques

### Fonction utiliser pour calculer le temps CPU

statistics(runtime,[Start,\_]),initial\_state(U), heuristique2(U,H),

statistics(runtime,[Stop,\_]),Runtime is Stop-Start.

### Heuristique 1

On a ici un runtime de 0. Le logiciel utilisé n’est pas assez précis au niveau du temps, la commande s’exécute trop rapidement.

Runtime de 0 sous swish pour h2 = 16

### Heuristique 2

Cette heuristique est basée sur la distance de Manhattan, pour chaque pièce on calcule la distance minimale à parcourir pour rejoindre sa position finale.

**Comparativement à l’heuristique 1, l’heuristique 2 est plus précise (H = 5 pour l'heuristique 2, contre H = 4 contre l’heuristique 1) alors que son temps d’exécution est deux fois plus lent.**

## Extensions prévues ou entrevues

On pourrait adapter notre code sans trop de problèmes à des taquin plus importants.

Si nous voulions appliquer A\* a un rubik’s cube, les lettres seraient remplacées par des couleurs. La matrice utilisables devraient être une matrice 12\*9 avec des cases inaccessibles pour les zones extérieurs à notre cube. Notre solution finale serait modélisé par l’image si dessous ou par 6 matrices de même couleurs juxtaposées.

# Rapport IA – TP2

Familiarisation avec le problème TicTacToe

1.2) ?- situation\_initiale(S), joueur\_initial(J).

Ici, S représente notre matrice du jeu à son état initial, c’est à dire vide. Cette matrice fait 3\*3. J représente le premier joueur.

?-situation\_initiale(S), nth1(3,S,Lig), nth1(2,Lig,o).

Ici on initialise notre matrice S, puis on se place sur la 3ème ligne de notre matrice S. Enfin, on place le jeton « o » sur cette ligne, colonne 2.

2.2) Pour gagner ce jeu, il faut avoir aligner 3 jetons sur une même ligne, colonne ou diagonale. Nous devons donc définir les termes d’un alignement gagnant :

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

On définit le prédicat ligne qui est vrai quand L fait partie de M. On définit ensuite une liste de prédicat jusqu’à seconde\_diag qui nous permettent de déterminer chaque type d’alignement existant dans une matrice NxN.

**Tests Unitaires :**

?- 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

On définit ensuite le prédicat unifiable, qui consiste à vérifier le caractère unifiable de chaque emplacement de la liste. Ainsi, si l’emplacement comporte le caractère \_, le prédicat retourne true, cela signifie que la place est libre.

unifiable(X,\_):-var(X).

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

**Tests unitaires :**

On vérifie ensuite un alignement possible pour un joueur grâce au prédicat possible :

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

possible( [], \_).

**Tests Unitaires :**

?- A=[\_,\_,\_], possible(A,x).

Yes

?- A=[x,\_,x], possible(A,x).

Yes

?- A=[\_,o,x], possible(A,x).

No

Enfin, on code les fonctions alignement gagnant et alignement perdant. Un alignement gagnant pour J est alignement possible pour J qui ne comporte pas d’espace libre. Un alignement perdant est un alignement gagnant pour son adversaire.

Code

Il faut ensuite modéliser la situation de la matrice lorsqu’un joueur joue et place un jeton sur la case de coordonnées [L,C]. On peut réaliser ça en codant successeur.