# Rapport de TP – Baures - Baudoint

Lien vers le git :

https://github.com/VBaures/IA.git

|  |
| --- |
| Table des matières  [Rapport de TP – Baures - Baudoint 1](#_Toc66636916)  [Table des matières 1](#_Toc66636917)  [Rapport IA – TP1 1](#_Toc66636918)  [Familiarisation avec le problème du Taquin 3\*3 1](#_Toc66636919)  [Développement des deux heuristiques 2](#_Toc66636920)  [Fonction utiliser pour calculer le temps CPU 2](#_Toc66636921)  [Heuristique 1 3](#_Toc66636922)  [Heuristique 2 3](#_Toc66636923)  [Implémentation de A\* 4](#_Toc66636924)  [Extensions prévues ou entrevues 5](#_Toc66636925)  [Rapport IA – TP2 5](#_Toc66636926)  [Familiarisation avec le problème TicTacToe 5](#_Toc66636927)  [Développement de l’algorithme Negamax 7](#_Toc66636928)  [Extension et développement 8](#_Toc66636929) Rapport IA – TP1Familiarisation 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.

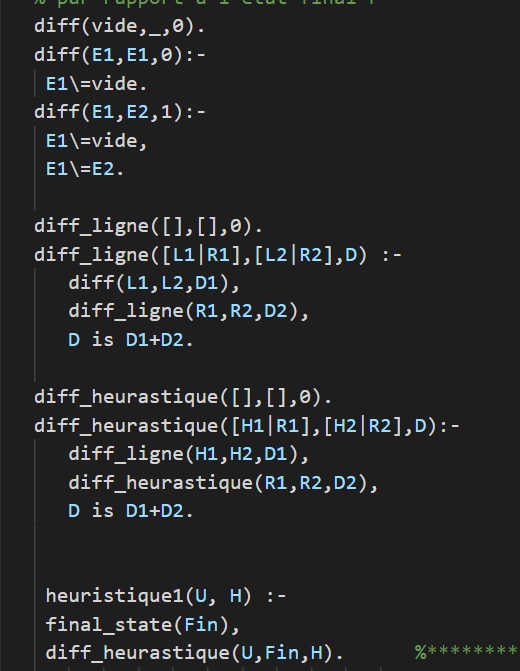
## Développement des deux heuristiques

### Fonction utiliser pour calculer le temps CPU

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

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

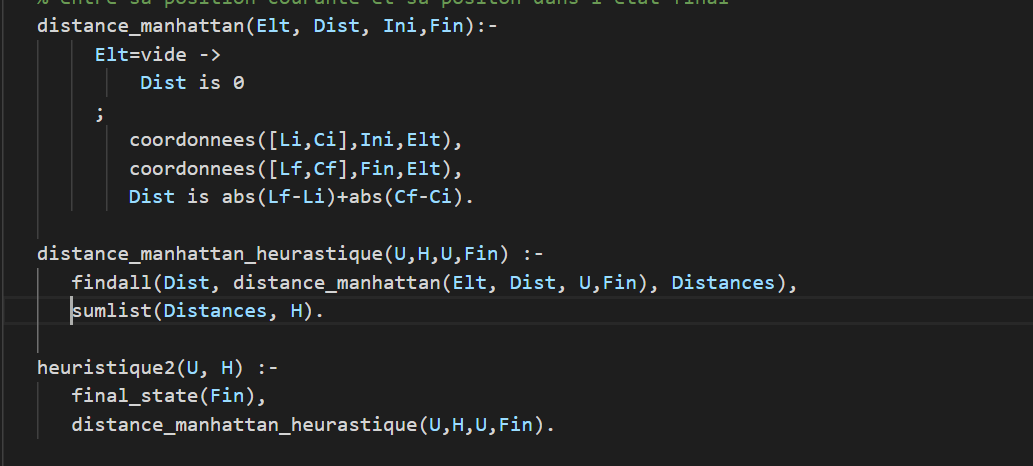
### Heuristique 1



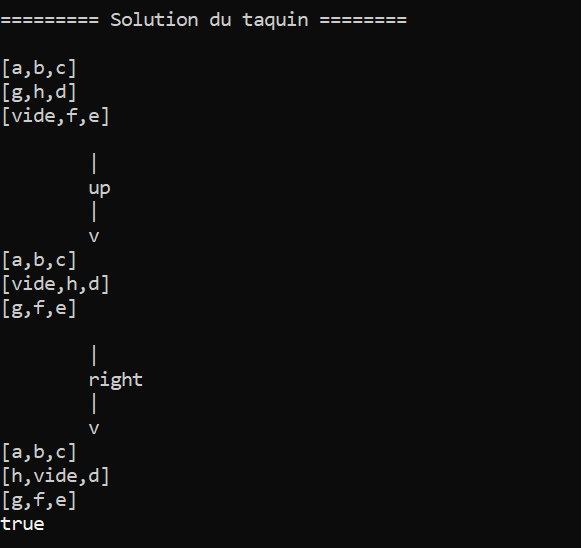
On a ici un runtime de 0. Le logiciel utilisé n’est pas assez précis au niveau du temps, la commande s’exécute trop rapidement.

### 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. Notre implémentation se fait en évaluant récursivement le nombre de différence entre les deux matrices. De même que pour le premier heuristique, le runtime est de 0.



## Implémentation de A\*



Solution pour le taquin avec 2 coups

|  |  |  |
| --- | --- | --- |
|  | **4 coups** | **10 coups** |
| **Temps Heuristique 1** | 0,001900 | 0,08745 |
| **Temps Heuristique 2** | 0,003354 | 0,01345 |

L’heuristique 2 est plus rapide que l’heuristique 1 à partir du moment où le nombre de coup est supérieur à 4.

La séquence maximale possible pour l’heuristique 1 est de 30.

## Extensions prévues ou entrevues

On pourrait adapter notre code sans trop de problèmes à des taquin plus importants ( 4\*4) en changeant simplement la matrice de départ et d’arrivée ( en les agrandissant ) , sans avoir à changer l’heuristique ni la logique de l’algorithme A\*.

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 :**

?- Unifiable(\_,x).

True

?- Unifiable(x,x)

True

?- Unifiable (o,x)

False

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.

heuristique(J,Situation,H) :-

    findall(A,alignement(A,Situation),Alignements),

    nb\_ligne\_possible(J,Alignements,Nperso),

    adversaire(J,Jadvers),

    nb\_ligne\_possible(Jadvers,Alignements,Nadvers),

    H is Nperso-Nadvers.

**Tests unitaires :**

?- joueur\_initiale(J), situation\_initiale(Situation), heuristique(J,Situation,H) .

J=x,

Situation = [[\_,\_,\_],[\_,\_,\_],[\_,\_,\_]]

H = 0

## Développement de l’algorithme Negamax

Il faut d’abord 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.

?- joueur\_initiale(J), situation\_initiale(Situation),successeur(J,Situation,[2,2]).

J=x,

Situation = [[\_,\_,\_],[\_,x,\_],[\_,\_,\_]]

On teste ensuite la fonction situation terminale développée. On regarde si on est dans une situation terminale ou non :

?- Situation=[[x,o,x][o,x,o],[x,o,x]], situation\_terminale(J,Situation).

Situation =[[x,o,x][o,x,o],[x,o,x]].

?- Situation=[[x,\_,x][o,x,o],[x,o,x]], situation\_terminale(J,Situation).

False.

On cherche ensuite quel couple à le coup le plus bas dans une liste grâce au prédicat meilleur.

?- meilleur([« a »,2],[« b »,4],[« c »,1]],Meilleur).

Meilleur = [« c »,1].

Test de notre algorithme negamax et report en temps CPU :

|  |  |  |  |
| --- | --- | --- | --- |
| **Pmax** | **CPU\_TIME** | **B** | **V** |
| 1 | 0 ,001486333 | [2,2] | 4 |
| 2 | 0,0068427189 | [2,2] | 1 |
| 3 | 0,0199972895 | [2,2] | 3 |
| 4 | 0,1062378089 | [2,2] | 1 |
| 5 | 0,5112175169 | [2,2] | 3 |
| 6 | 1,8448569033 | [2,2] | 1 |
| 7 | 5,688088772 | [2,2] | 2 |
| 8 | 11,296184598 | [3,3] | 1000 |
| 9 | 11,959933559 | [3,3] | 0 |

## Extension et développement

4.1) Pour 9, c’est la situation finale quand tout a été joué, c’est donc notre situation réelle., plus personne ne peut gagner.

4.2) On peut stocker nos états déjà traiter et faire tourner notre état afin de voir si il correspond à un état déjà traiter dans le tableau.

4.3)

* Il faut déjà reprendre la matrice de départ qui ne fera plus 3\*3 mais 6\*7.
* Il existe désormais 69 alignements possibles. Par colonnes on a 3\*7 possibilités, 4\*6 par lignes et enfin par diagonale on compte 12 diagonales montantes et 12 diagonales descendantes. Finalement on a 69 possibilités.