***Delahaye Chloé Master 1 IRCOMS***

***Teboul Nathan***

Compte rendu TP IA

-

Taquin

***Sommaire***

Table des matières

[Introduction : 3](#_Toc534882734)

[Etape préliminaire : Génération de taquin 4](#_Toc534882735)

[1) La classe Taquin 4](#_Toc534882736)

[2) Génération aléatoire de Taquin 4](#_Toc534882737)

[3) Vérification de la solubilité d’un taquin 5](#_Toc534882738)

[4) Heuristiques 7](#_Toc534882739)

[Etape 1 : Taquin 3x3 8](#_Toc534882740)

[1) But 8](#_Toc534882741)

[2) Algorithme A\* 8](#_Toc534882742)

[3) Algorithme recherche en cout uniforme 13](#_Toc534882743)

[Etape 2 : Taquin 4x4 16](#_Toc534882744)

[1) But 16](#_Toc534882745)

[2) Algorithme A\* 16](#_Toc534882746)

[3) Algorithme IDA \* 16](#_Toc534882747)

[4) Heuristiques 17](#_Toc534882748)

[Etape 3 : Taquins partiellement spécifiés 18](#_Toc534882749)

[1) But 18](#_Toc534882750)

[2) Réalisation 18](#_Toc534882751)

[Etape 4 : Résolution taquins 4x4 avec heuristiques 19](#_Toc534882752)

[1) But 19](#_Toc534882753)

[2) Exécution pour des taquins 4x4 19](#_Toc534882754)

[3) Conclusion 21](#_Toc534882755)

# Introduction :

L’objectif consiste à déterminer le nombre minimal de déplacements, nécessaire pour passer d’un taquin quelconque vers un taquin solution. L’objectif final sera de proposer l’étude des taquins 4 × 4.

L’étude se fera en 4 étapes :

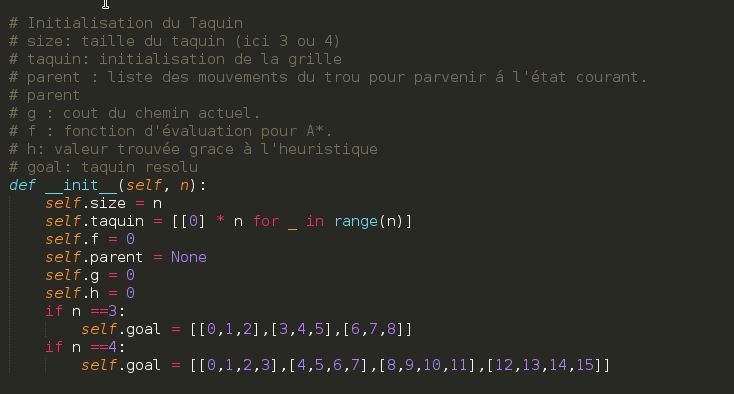
* Taquin 3x3 avec les algorithmes A\* et l’algorithme de recherche en couts uniformes
* Taquin 4x4 avec les algorithmes A\* et IDA\*
* Taquin partiellement spécifiés avec les algorithmes A∗ et IDA∗
* Résolution taquin 4x4 avec les 3 heuristiques proposées (h0 : la valeur associée à chaque état est égale à 0, h1 : pièces mal positionnées, h2 : distance de Manhattan)

# Etape préliminaire : Génération de taquin

## La classe Taquin

On initialise le taquin grâce à la fonction init. On passe en paramètre la taille du taquin.

Les différents attributs et leur signification sont expliqués dans les commentaires du code dans la capture d’écran ci-dessous.



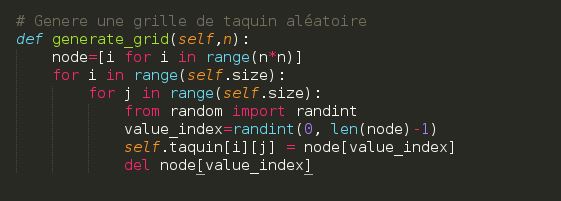
## Génération aléatoire de Taquin

Deux solutions s’offraient à nous quant à la génération d’un Taquin, car Parmi toutes les dispositions initiales (pour un taquin 4x4), il existe 10 461 394 944 000 dispositions dont la résolution est possible (à savoir la moitié de la [factorielle](https://fr.wikipedia.org/wiki/Factorielle) de 16), et autant d'impossibles :

* Partir d’un taquin solution, le mélanger pour obtenir un taquin de départ : ainsi nous sommes sûr que le taquin peut être résolu.
* Générer un taquin aléatoirement, et créer une fonction qui vérifiera avant la résolution si le taquin peut effectivement être résolu ou non.

Pour ce TP, nous avons choisi la 2e solution.

Nous passons en paramètre la taille du taquin afin de générer un taquin de la bonne taille. Le tableau node prendra les chiffres de 0 à 8 (pour n=3) ou de 0 à 15 (pour n=4), on pourra ainsi « piocher » dans ce tableau les chiffres qui permettront de réaliser le taquin initial aléatoire. On parcourt ensuite le taquin remplit de zéro en remplaçant les zéros par les valeurs adéquat en piochant aléatoirement dans la liste node. On obtient ainsi le taquin initial aléatoire.



## Vérification de la solubilité d’un taquin

De nombreux taquins ne peuvent pas être résolus. Mais il est possible de dire à l'avance si le problème posé est soluble ou non. En effet, la configuration initiale d'un taquin est une [permutation](https://fr.wikipedia.org/wiki/Permutation) de sa configuration finale.

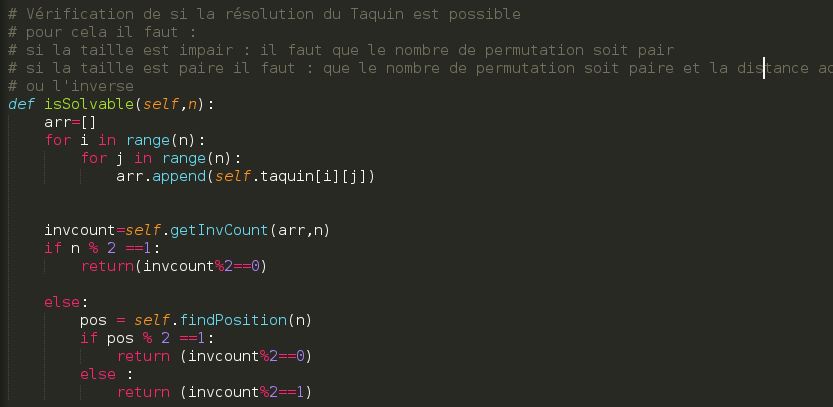
En général, pour une grille de taille n, on peut trouver si un taquin n\*n peut être résolu ou non en suivant les règles suivantes :

1. Si n est impair, alors le taquin peut être résolu si le nombre de permutations est pair dans l’état initial.
2. Si n est pair, la taquin peut être résolu si

* La case vide est dans une ligne paire (en comptant à partir de la fin) et le nombre de permutations est impair.
* La case vide est dans une ligne impaire (en comptant à partir de la fin) et le nombre de permutations est pair.
* Pour les autres cas il ne peut pas être résolu.

Premièrement nous avons créé une fonction (getInvCount) permettant de compter le nombre de permutations dans un taquin passé en paramètre. Ensuite nous créons une fonction (findPosition) qui va chercher la position de la case vide. Ces deux fonctions vont être appelées dans la fonction isSolvable, qui va vérifier les conditions énoncées précédemment afin de retourner si le taquin peut être résolu ou non.





Exemple d’exécution de la fonction :





## Heuristiques

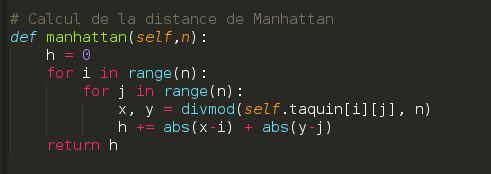
1. Cases mal placées

La fonction suivante permet de déterminer le nombre de cases mal placées dans un taquin intermédiaire par rapport au taquin solution.



1. Distance de Manhattan

La fonction suivante permet de déterminer la distance de Manhattan dans un taquin intermédiaire.



# Etape 1 : Taquin 3x3

## But

Nous allons dans un premier temps considérer les taquins 3 × 3 (Tableau 21). Le travail demandé est de programmer A∗ et l’algorithme de recherche en coût uniforme. On utilise les heuristiques suivantes :

* h0 : la valeur associée à chaque état est égale à 0
* h1 : pièces mal positionnées
* h2 : distance de Manhattan

## Algorithme A\*

1. Algorithme

L’algorithme de recherche A\* est un algorithme de recherche de chemin dans un graphe entre un nœud initial et un nœud final donnés.

A chaque itération, A\* va tenter de prendre le chemin le plus court pour aller d’une source à une destination.

On travaille à partir d’une position de départ (ici start), d’une arrivée (ici current.goal sous entendue lors de l’appel de current.is\_solution()), d’un plateau (ici current.taquin), et de deux listes de nœuds (ouverte, ici openList et fermée, ici closeList), vides au départ.

A partir du départ, on regarde quels sont les nœuds voisins (correspondant aux mouvements possibles, ici grâce à la fonction move\_function()). On place le nœud de départ dans la liste fermée.

Pour chaque nœud voisin (ici récupérés dans la variable X) :

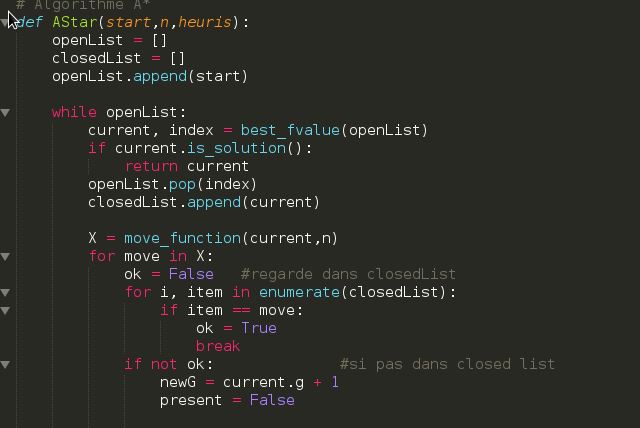
* S’il est déjà dans closeList, on passe car on a déjà étudié ce trajet
* S’il est déjà dans openList, on vérifie sa valeur. Si la valeur est plus petite, c’est que le chemin est plus court en passant par le trajet courant, donc on met à jour sa valeur dans openList, et on met à jour son parent (avec le nœud courant qui correspond à un meilleur chemin)
* Sinon on ajoute ce nœud à openList avec comme parent le nœud courant

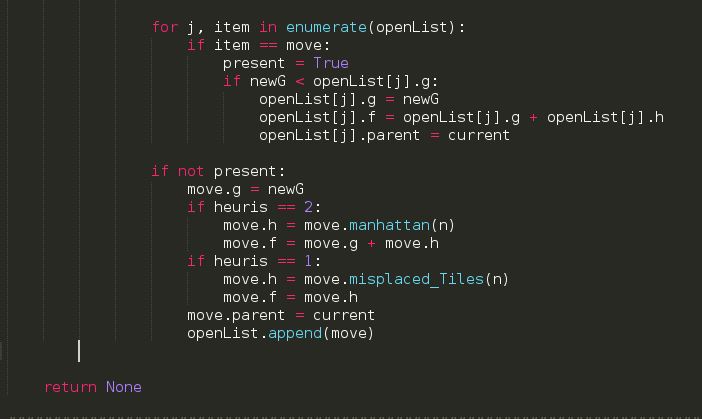
On parcourt openList à la recherche du nœud ayant plus petite valeur.

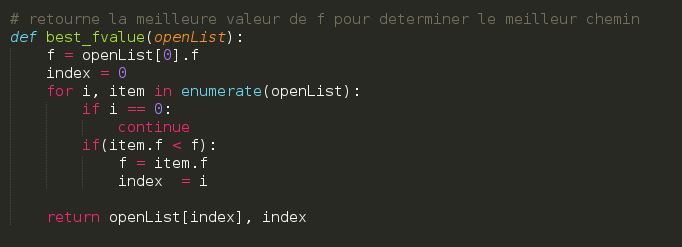
Si openList est vide, il n’y a pas de solution.

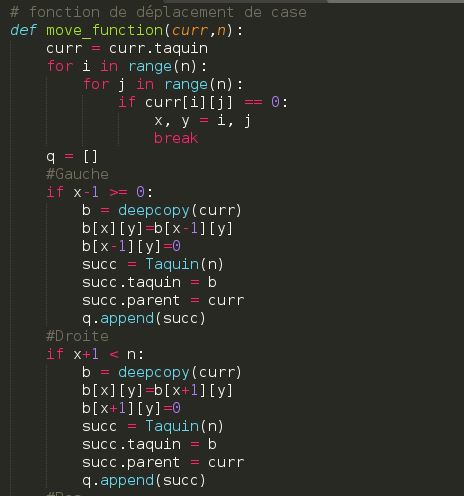
On prend le meilleur nœud, et on le retire de openList pour le mettre dans closeList.

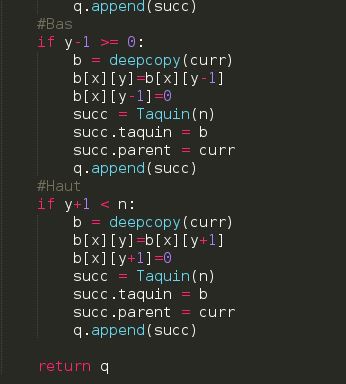
Puis on vérifie si le nœud courant est destination, alors on a trouvé un bon chemin.





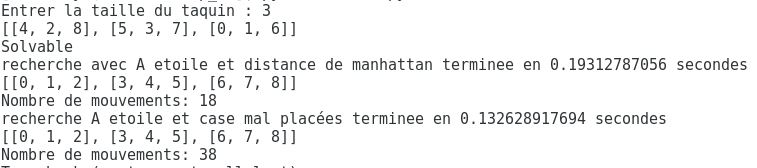




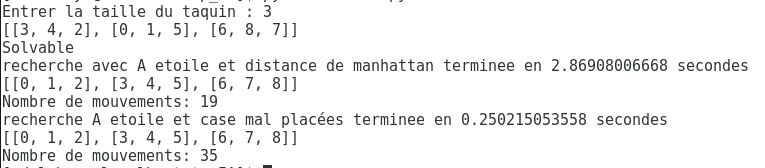


1. Mesure du temps d’exécution

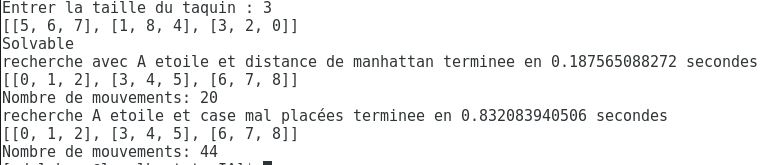
Taquin 1 :



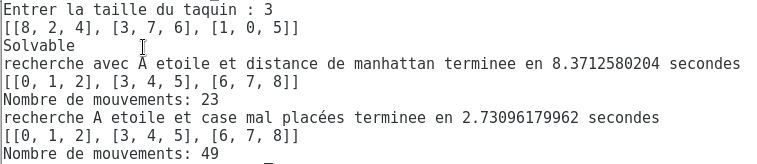
Taquin 2 :



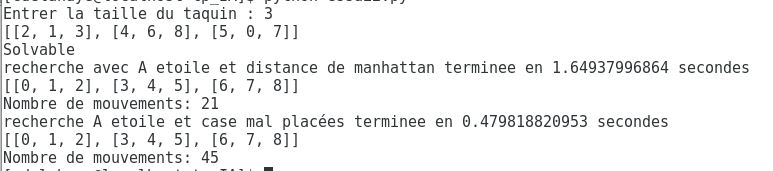
Taquin 3 :



Taquin 4 :



Taquin 5 :



|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **Taquin 1** | **Taquin 2** | **Taquin 3** | **Taquin 4** | **Taquin 5** |
| **A\* distance de Manhattan/ Temps d’exécution (s)** | 0.19312 | 2.8690 | 0.1875 | 8.3712 | 1.6493 |
| **A\* distance de Manhattan/ Nombre de déplacements** | 18 | 19 | 20 | 23 | 21 |
| **A\* cases mal placées/ Temps d’exécution (s)** | 0.1326 | 0.2502 | 0.8320 | 2.7309 | 0.4798 |
| **A\* cases mal placées/ Nombre de déplacements** | 38 | 35 | 44 | 49 | 45 |

On constate que de manière générale, pour les taquins générés ci-dessus, l’heuristique « cases mal placées » (h1) est plus efficace en termes de temps d’exécution que la distance de Manhattan (h2), mais cette constatation dépend du taquin généré et n’est pas vrai dans 100% des cas (voir Taquin 3 dans les exemples d’exécutions ci-dessus).

On peut également constater que plus le taquin est difficile (ie. Plus il va falloir de temps/ de mouvements pour le résoudre), plus la différence de temps d’exécution est significative (jusqu’à plusieurs secondes d’écart).

Malgré cela, l’heuristique h2 est très efficace en terme de nombres de mouvement pour arriver à la solution, car elle permet de diminuer le nombre de déplacements nécessaires (de plus de la moitié dans les exemples ci-dessus).

## Algorithme recherche en cout uniforme

1. Algorithme

L’algorithme de recherche est le meilleur algorithme de recherche n’impliquant pas d’heuristique, il s’agit d’un cas particulier de l’algorithme de recherche en largeur (breadth search first). Au lieu de considérer les nœuds dans l’ordre de leur profondeur depuis la racine, on considère les couts depuis la racine.

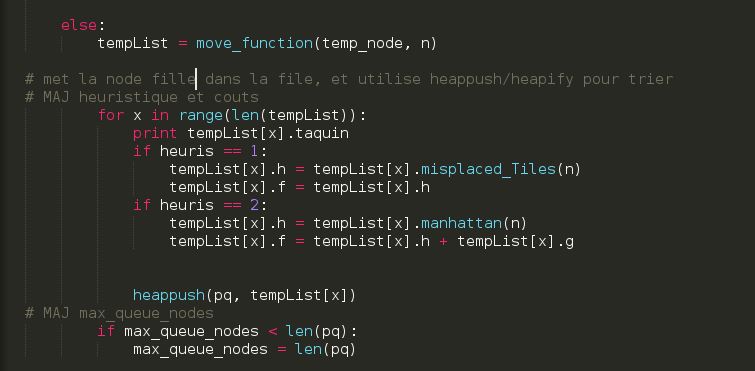
On utilise une priority queue (ici pq). A chaque étape, les nœuds sont insérés dans la file (ici heappush(pq,templList)) ordonnée dans l`ordre croissant de la fonction *g(n)* (qui calcule les couts), de façon qu`à chaque itération le nœud qui minimise *g(n)* soit développé (ici grâce à move\_function) pour obtenir ainsi un chemin de coût minimum entre l’état initial et la solution.

On choisit un nœud dans la file, si ce nœud est solution, le travail est terminé et on peut retourner le chemin trouvé.

Sinon on supprime ce nœud de la file et on recherche ses successeurs. Pour chaque successeur de ce nœud, on calcule sa valeur et on l’ajoute à pq.

L’algorithme est complet et optimal, mais a une efficacité réduite.





1. Mesure du temps d’exécution

Nous n’avons malheureusement pas de résultats permettant de faire l’analyse de l’efficacité de cet algorithme, car le fait de ne pas utiliser d’heuristiques rend la recherche de solution beaucoup trop longue.

# Etape 2 : Taquin 4x4

## But

Le but est de proposer une adaptation de notre algorithme A∗ pour des taquins 4 × 4. On développera également l’algorithme IDA∗.

Nous allons aussi chercher une autre heuristique possible pour la réalisation du solveur de taquin 4x4 et expliquer si cette heuristique et intéressante pour notre problème ou non.

## Algorithme A\*

L’algorithme A\* utilisé pour la résolution du taquin 4x4 est le même que pour le taquin 3x3, n étant passé en paramètre et la fonction créée en fonction de n.

## Algorithme IDA \*

L’algorithme IDA\* (Iterative deepening A\*) est une variation de l’algorithme A\* qui utilise l’algorithme de recherche en profondeur.

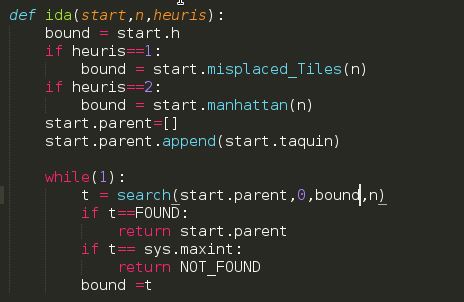
IDA\* utilise comme limite de l'horizon de recherche la valeur de la fonction heuristique. Chaque itération de IDA\* est une recherche en profondeur qui étend tous les nœuds à l'intérieur d'un contour délimité par la valeur courante de flimite en regardant par-dessus cette limite pour savoir où se trouvera la prochaine valeur de flimite.

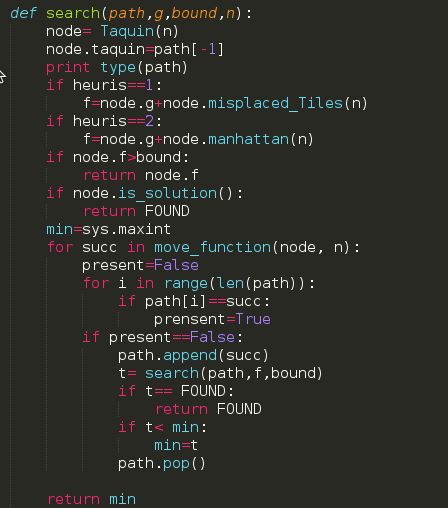
Ici la limite est calculée en fonction de l’heuristique, puis tant que la solution n’est pas trouvée, on utilise l’algorithme de recherche en profondeur (ici search()) avec un limite différente.

C'est un algorithme de recherche qui progresse à partir d'un sommet (ici node) en s'appelant récursivement pour chaque sommet voisin ce sommet (voisin trouvés grâce à move\_function().

Pour chaque sommet, il marque le sommet actuel, et il prend le premier sommet voisin jusqu'à ce qu'un sommet n'ait plus de voisins (ou que tous ses voisins soient marqués), et revient alors au sommet père (ici boucle for succ in move\_funtion()).

L'algorithme pourrait tourner indéfiniment, c'est pour cela que l'on doit en outre marquer chaque sommet déjà parcouru, et ne parcourir que les sommets qui ne sont pas encore marqués.





## Heuristiques

Une autre heuristique possible serait la distance euclidienne, car elle peut être utilisée quel que soit les directions et le nombre de directions possibles pour les déplacements. Il s’agit de calculer la distance entre la cellule courante et la cellule but en utilisant la formule de calcul de distance : √((currentcell.x-goalcell.x)2+ (currentcell.y-goalcell.y)2).

Par contre, même si elle peut être utilisée comme heuristique pour A\*, et que la distance euclidienne est plus courte que la distance de Manhattan (et donc donnera toujours les plus courts chemins), elle va rendre l’exécution de l’algorithme A\* beaucoup plus longue.

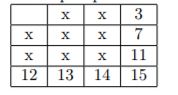
# Etape 3 : Taquins partiellement spécifiés

## But

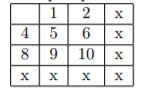
Un taquin partiellement spécifié est un taquin dont les cases numérotées et la case vide sont spécifiées (la case représentée par un ’x’ est non spécifiée). Ce taquin est appelé frange (frontière ou encore fringe).

Le but est de calculer le coût réel nécessaire en utilisant les algorithmes A\* et IDA\* :

* Pour passer d’une permutation quelconque d’un taquin frange (un taquin 4x4 ordinaire) vers le taquin frange suivant :



* Pour passer d’une permutation quelconque d’un taquin frange (un taquin 4x4 ordinaire) vers un autre taquin donné ci-dessous :



## Réalisation

Nous n’avons malheureusement pas eu le temps d’implémenter cette étape du TP.

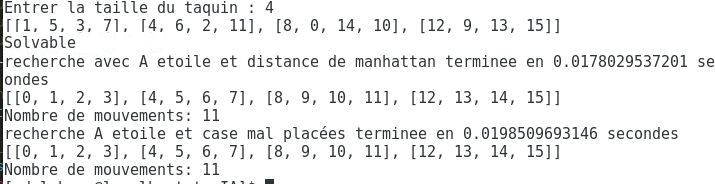
# Etape 4 : Résolution taquins 4x4 avec heuristiques

## But

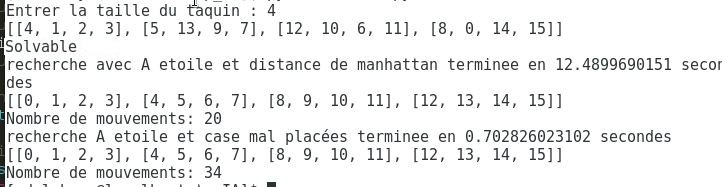
Résolution des taquins. On utilisera les heuristiques h0, h1 et h2 définies précédemment.

## Exécution pour des taquins 4x4

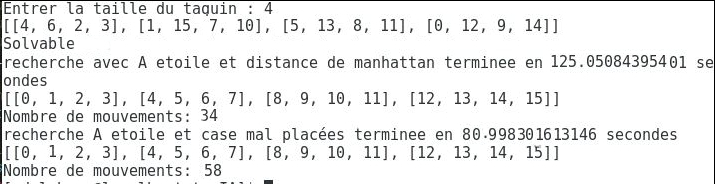
Taquin 1 :

******

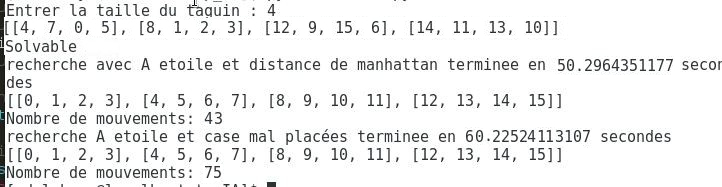
Taquin 2 :



Taquin 3 :

******

Taquin 4 :

******

## Conclusion

En ce qui concerne le temps d’exécution, il nous est difficile de conclure sur l’efficacité des heuristiques. En effet, l’efficacité en termes de temps d’exécution est très dépendante du taquin généré, rendant tantôt h1 plus efficace que h2 et inversement.

Cependant, comme pour les taquin 3x3, on remarque toujours que les différences de temps d’exécution sont toujours de plus en plus significatives à mesure que la difficulté du taquin augmente.

Mais, de la même façon que pour les taquin 3x3, nous pouvons toujours conclure que l’heuristique h2 (distance de Manhattan) est la plus efficace en termes de nombre de déplacements, en effet quelque soit le taquin généré, le nombre de déplacement est beaucoup plus petits pour l’heuristique « distance de Manhattan » que pour l’heuristique « cases mal placées ».