***Delahaye Chloé Master 1 IRCOMS***

***Teboul Nathan***

Compte rendu TP IA

-

Taquin

***Sommaire***

Table des matières

[Introduction : 3](#_Toc534576760)

[Etape préliminaire : Génération de taquin 4](#_Toc534576761)

[1) La classe Taquin 4](#_Toc534576762)

[2) Génération aléatoire de Taquin 4](#_Toc534576763)

[3) Vérification de la solubilité d’un taquin 5](#_Toc534576764)

[4) Heuristiques 6](#_Toc534576765)

[Etape 1 : Taquin 3x3 8](#_Toc534576766)

[1) Algorithme A\* 8](#_Toc534576767)

[2) Algorithme recherche en cout uniforme 8](#_Toc534576768)

[Etape 2 : Taquin 4x4 8](#_Toc534576769)

[1) Algorithme A\* 9](#_Toc534576770)

[2) Algorithme IDA \* 9](#_Toc534576771)

[3) Heuristiques 9](#_Toc534576772)

[Etape 3 : Taquins partiellement spécifiés 10](#_Toc534576773)

[1) Algorithme A\* 10](#_Toc534576774)

[2) Algorithme IDA\* 10](#_Toc534576775)

[Etape 4 : Résolution taquins 4x4 avec heuristiques 10](#_Toc534576776)

[Conclusion 10](#_Toc534576777)

# 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 :

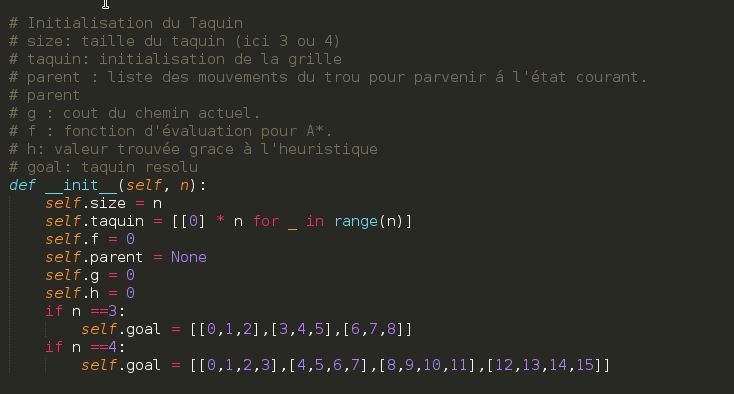
* Taquin 3x3 avec les algorithmes A\* et l’algorithme de recherche en couts uniformes
* Taquin 4x4 avec les algorithmes A\* et IDA\*
* Taquin partiellement spécifiés avec les algorithmes A∗ et IDA∗
* Résolution taquin 4x4 avec les 3 heuristiques proposées (h0 : la valeur associée à chaque état est égale 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

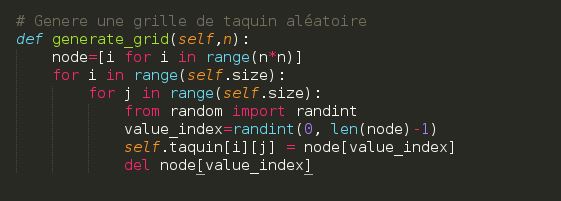
Deux solutions s’offraient à nous quant à la génération d’un Taquin, car Parmi toutes les dispositions initiales (pour un taquin 4x4), il existe 10 461 394 944 000 dispositions dont la résolution est possible (à savoir la moitié de la [factorielle](https://fr.wikipedia.org/wiki/Factorielle) de 16), et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

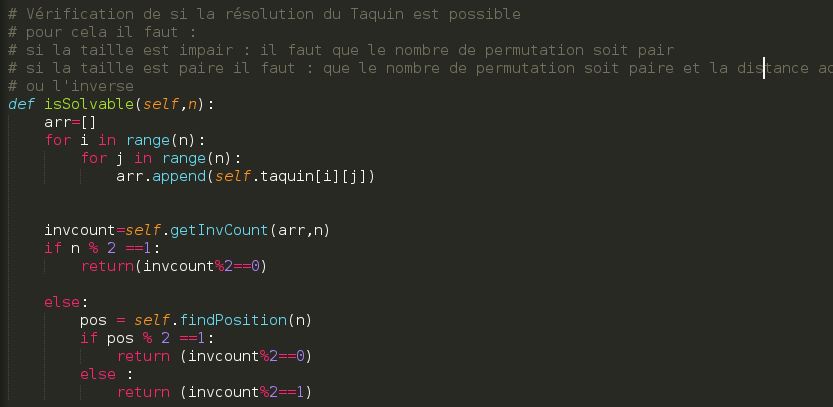
En général, pour une grille de taille n, on peut trouver si un taquin n\*n peut être résolu ou non en suivant les règles suivantes :

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

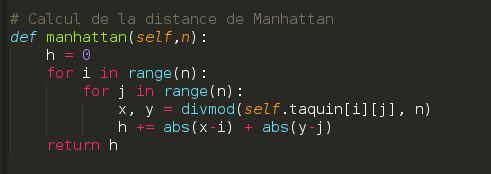
1. Cases mal placées

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

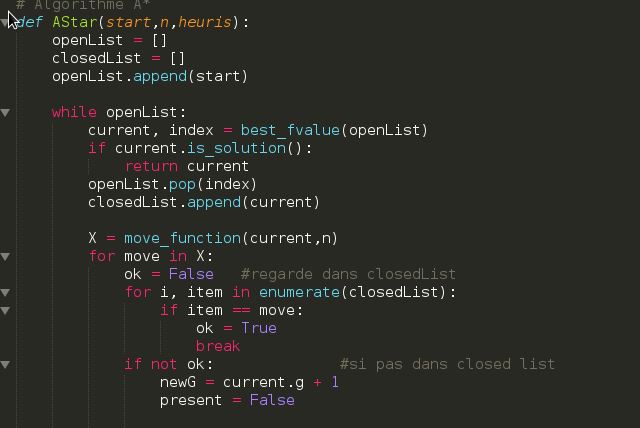
La fonction suivante permet de déterminer la distance de Manhattan dans un taquin intermédiaire.

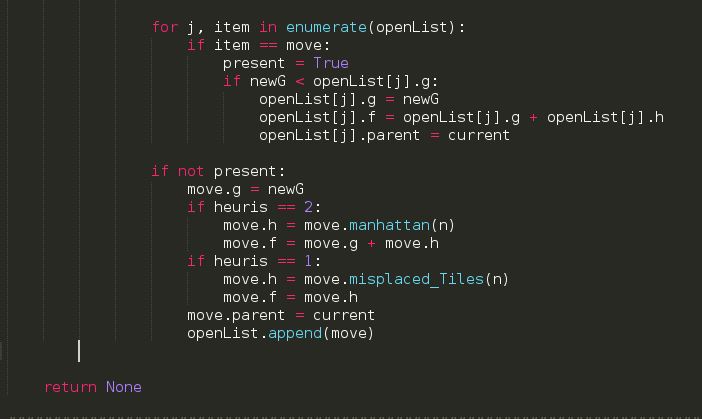


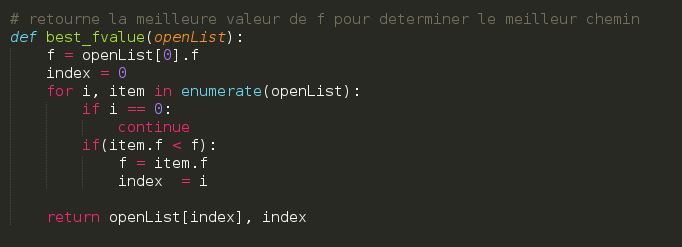
# Etape 1 : Taquin 3x3

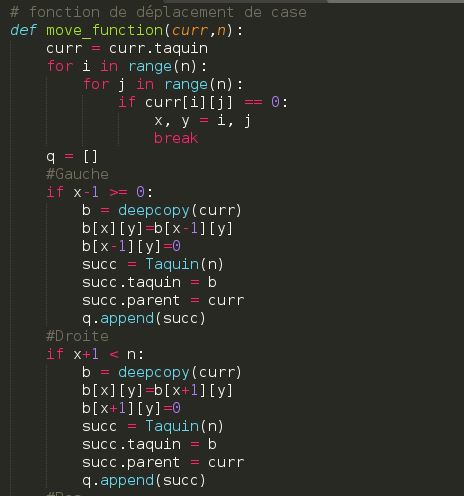
## Algorithme A\*

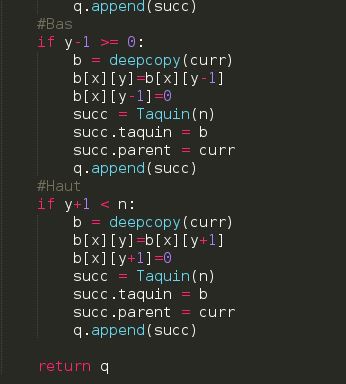
1. Algorithme











1. Mesure du temps d’exécution

## Algorithme recherche en cout uniforme

1. Algorithme
2. Mesure du temps d’exécution

# Etape 2 : Taquin 4x4

## Algorithme A\*

1. Algorithme

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.

1. Mesure du temps d’exécution

## Algorithme IDA \*

1. Algorithme
2. Mesure du temps d’exécution

## Heuristiques

# Etape 3 : Taquins partiellement spécifiés

## Algorithme A\*

## Algorithme IDA\*

# Etape 4 : Résolution taquins 4x4 avec heuristiques

# Conclusion