

Le Taquin

Mélanie MARQUES & Guillaume COQUARD

Rendu le 18 mars 2019

Chapitre 1

Etude Théorique

1.1 Le Problème du Taquin

Un taquin $n \times n$ est un puzzle carré, d'une largeur l et d'une taille t , telles que $l = n$, $t = n \times n$, composé de $t - 1$ tuiles numérotées de 1 à t et d'un trou.



Figure 1.1 – Un départ possible et l'arrivée voulue

Les tuiles ne peuvent se déplacer que par glissement dans la seule case vide à un moment donné. Le jeu consiste à replacer les tuiles dans l'ordre numérique. Ainsi, par le biais de ce projet, nous allons donc, à l'aide du langage de programmation Python, développer ce jeu et répondre aux deux questions suivantes : Quelle est la séquence minimale de mouvements à faire sur un taquin pour obtenir la solution ? Comment trouver cette séquence ?

1.2 Etude du cas général

Pour ce faire, nous détaillerons dans cette partie l'algorithme A^* , prononcer A star, utilisé pour résoudre le problème, la définition des états et des actions que l'agent pourra réaliser

dans chaque état puis nous aborderons spécifiquement la stratégie de recherche utilisée par l'algorithme dans le but d'obtenir une solution optimale.

1.2.1 Définition d'un état

Afin de suivre précisément l'évolution de la résolution du problème, il convient de formaliser tout d'abord ce qu'est un état, et l'environnement dans lequel il évolue. Ainsi, par l'intermédiaire de la programmation orientée objet, nous pouvons définir l'environnement et les états par les attributs suivants :

Environnement

sizes : les dimensions du taquin, largeur et taille

choices : les heuristiques choisies pour une exécution, soient les identifiants de chaque pondération ou de l'heuristique de la Mauvaise Place par exemple

weightings : les pondérations utilisées pour une exécution

moves : l'historique des coups joués par l'utilisateur

end : la solution trouvée par l'algorithme

Taquin

environment : l'environnement dans lequel évolue l'état

previous : la référence au taquin précédent – parent

sequence : l'ordre des tuiles dans le taquin, représenté par une liste, le vide vaut 0 ; à l'état initial la sequence est une liste remplie aléatoirement

inv : le nombre d'inversions, c'est-à-dire, le nombre de fois pour chaque élément de la sequence où celui-ci est plus grand que chacun des éléments suivants

dis : l'abréviation de l'anglais *disorder*, désordre, soit le nombre de tuiles qui ne sont pas à la place occupée dans l'état final

man : la distance de Manhattan brute, n'ayant subie aucune pondération

path : le chemin emprunté par l'algorithme pour atteindre l'état actuel, représenté par une suite de lettres : **L** pour gauche (left), **R** pour droite (right), **U** pour haut (up) et **D** pour bas (down)

moves : la liste des prochains coups possibles à partir de l'état actuel

h : la somme des calculs de chaque heuristique utilisée

g : le coût d'un chemin allant de l'état initial à l'état actuel représenté par un entier

f : la fonction d'évaluation : $f(n) = g(n) + h(n)$ avec n le taquin actuel

1.2.2 Détail des heuristiques

Dans le cadre de l'utilisation de l'algorithme A^* , le choix d'heuristiques est nécessaire. De fait, nous aurons recours à 7 heuristiques, surestimant la longueur du chemin à parcourir pour atteindre l'état final. Les 6 premières sont des distances dérivées de la distance de

Manhattan, correspondant aux heuristiques **H.1, H.2, H.3, H.4, H.5, H.6** et pondérées par les jeux de poids suivants :

	1	2	3	4	5	6	7	8
π_1	36	12	12	4	1	1	4	1
$\pi_2 = \pi_3$	8	7	6	5	4	3	2	1
$\pi_4 = \pi_5$	8	7	6	5	3	2	4	1
π_6	1	1	1	1	1	1	1	1

Figure 1.2 – Tableau des pondérations des distances de Manhattan

A partir de ces pondérations nous pouvons calculer une distance de Manhattan grâce à la formule suivante :

$$h_k(E) = \left(\sum_{i=1}^8 \pi_k(i) \times \varepsilon_E(i) \right) \text{ div } \rho_k$$

$\varepsilon_E(i)$ correspond au nombre de déplacements élémentaires nécessaire pour déplacer la tuile de numéro i de sa position initiale à sa position dans l'état final. On calcule ce nombre grâce à la formule suivante :

$$d(A, B) = |X_B - X_A| + |Y_B - Y_A|$$

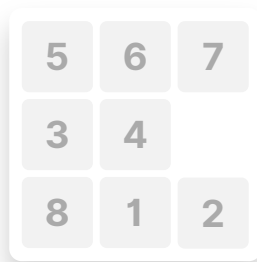
Sur le taquin ci-dessus, on a pour $\varepsilon_E(5)$:

- $x_{\text{Etat initial } 5} = 1$ $x_{\text{Etat Final } 5} = 2$
- $y_{\text{Etat initial } 5} = 1$ $y_{\text{Etat Final } 5} = 2$

Donc $\varepsilon_E(5) = 2$

Les coefficients de normalisation correspondant sont pour une heuristique i :

- $\rho_{i=1} = \rho_{i=3} = \rho_{i=5} = 4$
- $\rho_{i=2} = \rho_{i=4} = \rho_{i=6} = 1$



Compte tenu du choix effectué sur la définition d'un état pour un taquin de largeur 3 :

A l'état initial :

— Environnement :

sizes = [3, 9]

choices = [1, 2, 3, 4, 5, 6]

weightings = *Une liste des jeux de pondérations choisis*

moves = [Taquin Initial]

end = []

— Taquin Initial :

environment = *L'environnement ci-dessus*

previous = None

sequence = [4, 6, 2, 1, 3, 5, 0, 7, 8] *Une liste générée aléatoirement et valide*

inv = 8

dis = 8

man = 10

path = ""

moves = ["L", "D"]

h = 142

g = 0

f = 142

A l'état final :

— Taquin Final :

environment = *L'environnement est toujours le même, seul l'état final est ajouté à la liste end*

previous = *L'avant-dernier état*

sequence = [1, 2, 3, 4, 5, 6, 7, 8, 0]

inv = 0

dis = 0
man = 0
path = *Le chemin minimal depuis l'état initial jusqu'à cet état*
moves = ["R"]
h = 0
g = *La longueur du chemin minimal*
f = g

1.2.3 Définition d'une action

Une fois les états définis, il faut désormais s'intéresser aux actions. Une action dans ce problème est le déplacement d'une tuile, qui consiste en l'inversion de sa position avec celle du vide.

Seule les tuiles qui jouxtent le vide sont candidates à ces déplacements et une tuile qui vient d'être déplacée ne revient pas au coup d'après à sa position précédente pour éviter la possibilité de créer un cycle et ainsi accélérer le parcours des états. Ainsi on peut en tirer les conclusions suivantes :

- Si la tuile vide se trouve à l'intérieur du taquin à l'état initial, il y 4 coups possibles.
- Si la tuile vide se trouve sur un bord du taquin (mais pas dans un coin) du taquin à l'état initial, il y 3 coups possibles.
- Si la tuile vide se trouve dans un coin du taquin à l'état initial, il y 2 coups possibles.
- En revanche, si l'état actuel n'est pas l'état initial, il faut retirer 1 coup à chacune des valeurs précédentes.

1.2.4 Définition de l'agent

Il s'agit ici d'agents à réflexes simples : les agents agissent en tenant compte du dernier percept uniquement. Nous allons donc utiliser de règles de type action \Rightarrow condition.

1.2.5 Choix de l'algorithme

Après avoir défini les états et les actions, nous nous intéressons à la stratégie utilisée par l'algorithme de recherche A^* dans le but de trouver une séquence d'actions permettant d'atteindre l'état du but à partir de l'état initial.

La stratégie adoptée par l'algorithme consiste en la construction d'un arbre de recherche et d'y rechercher la solution. Dans ce sens, A^* va utiliser une fonction d'évaluation basée sur des heuristiques sur chaque nœud de l'arbre afin d'estimer le meilleur chemin à emprunter et d'en visiter les nœuds enfants, triés grâce à cette fonction d'évaluation.

Le principe repose sur le fait qu'à chaque itération, on tente de se rapprocher de la destination, c'est pourquoi on privilégie les états directement les plus proches de la destination en mettant de côté tous les autres.

Tous les états ne permettant pas de se rapprocher de la destination sont éloignés et placés plus loin dans la file d'exécution. Les états parcourus sont placés dans une liste des états explorés de manière à vérifier que le chemin actuellement emprunté est plus performant que celui stocké dans la liste.

Si cela s'avérait être le contraire, alors l'exploration actuelle s'arrêterait pour passer à l'exploration d'un autre chemin. Ainsi, on est sûr d'emprunter un chemin minimal pour les heuristiques choisies et d'éviter d'explorer à nouveau un même nœud mais de coût plus élevé. C'est l'association de ces différents principes qui assure la complétude de l'algorithme.

1.2.6 Implémentation de A^*

Le problème majeur de A^* est sa complexité exponentielle en espace mémoire. Tous les états créés sont conservés, ce qui peut selon les dimensions et l'état initial du taquin, rapidement saturer la mémoire de l'ordinateur, même avec les grandes capacités de mémoire actuelles.

Par conséquent, nous nous sommes, lors de la réalisation de notre programme, cherché à réduire le nombre d'états et à établir la meilleure gestion possible de ces derniers. En ce sens, nous avons pensé judicieux d'appliquer quelques optimisations :

Utiliser un dictionnaire ordonné afin de gérer la frontière En effet, la frontière peut comporter énormément d'états. C'est pour cela qu'il convient de trier ces états afin d'en permettre une meilleure gestion. Un dictionnaire ordonné en fonction d'une clef (la fonction d'évaluation f dans le cas présent) nous a paru le plus adapté. Cela nous permet d'éviter de parcourir la liste des états dans le but d'expanser celui possédant le f le plus petit. Grâce à cette structure, nous expansons uniquement l'état présent en premier dans le dictionnaire.

Utiliser un dictionnaire pour les états explorés Afin de réduire le temps de traitement et diminuer les expansions, il paraît logique de supprimer les états redondants et par conséquent inutiles. Ce que nous désignons par états redondants, sont en fait les états caractérisés par une séquence similaire mais un f différent. Ici, l'état qui va nous intéresser pour une même séquence est l'état possédant un f le plus petit possible.

C'est pourquoi notre dictionnaire va ordonner les états en fonction de la séquence (transformée en chaîne de caractères). Après la création des nœuds, nous vérifions que leur séquence n'est pas similaire à celle d'un état présent dans les états explorés. Si c'est le cas : nous regardons le f . Si le f de l'enfant est supérieur à celui de l'état exploré, nous supprimons l'enfant.

Dans cette optique, notre dictionnaire va prendre en clef la séquence (transformée en chaîne de caractères). Sinon, nous supprimons le l'état présent dans les états explorés. De même, lorsque nous expansons un état, nous l'ajoutons dans la liste des états explorés.

Cette structure de donnée nous permet en outre d'optimiser le temps de recherche d'un état exploré.

Chapitre 2

Etude Expérimentale

Dans cette première partie nous allons nous intéresser aux performances de l'algorithme sur des taquins 3*3. Pour cela nous exécuterons le programme sur les taquins suivants (solubles et générés aléatoirement par une fonction de notre programme) :

- [3, 4, 0, 7, 2, 1, 8, 6, 5]
- [5, 4, 3, 0, 7, 1, 2, 6, 8]
- [8, 4, 5, 7, 0, 6, 3, 1, 2]
- [0, 3, 8, 4, 6, 1, 7, 5, 2]
- [6, 7, 5, 4, 8, 2, 0, 3, 1]
- [6, 2, 5, 3, 4, 7, 1, 0, 8]
- [3, 6, 5, 0, 1, 4, 8, 2, 7]
- [3, 0, 6, 1, 8, 2, 5, 7, 4]
- [2, 5, 7, 0, 1, 6, 8, 3, 4]
- [0, 2, 8, 1, 4, 5, 6, 3, 7]

A partir de ces derniers, nous collecterons des informations relatives au temps d'exécution du programme, le nombre d'états créés, l'écart à la solution optimale ainsi que le taux d'optimalité (nombre de fois moyens où la solution renvoyée est minimale) le tout en faisant varier les heuristiques.

Comparaison des heuristiques individuellement Dans un premier temps, nous allons nous attacher à comparer les différentes heuristiques utilisées dans le but de résoudre ce problème.

En ce sens, nous avons exécuté 10 fois le programme en utilisant chaque heuristique afin de collecter des informations relatives au temps d'exécution du programme, le nombre d'états créés, et l'optimalité de la solution :

	Temps	Etats créés	Ecart à la Solution	Optimalité
H.1	26ms	624	5,2	$\frac{1}{5}$
H.2	18ms	450	9,2	$\frac{1}{5}$
H.3	49ms	1524	0,6	$\frac{2}{3}$
H.4	14ms	411	8,8	$\frac{1}{5}$
H.5	43ms	1342	0,6	$\frac{2}{3}$
H.6	87ms	2831	0	1
Désordre	1,2s	36089	0	1

Nous remarquons donc que H.4 présente le temps d'exécution moyen, et un nombre d'état créé le plus faible cependant elle ne renvoie un résultat optimal qu'une exécution sur 5.

Au contraire, les heuristiques H.6 et Désordre renvoient à chaque exécution un résultat optimal mais leurs temps d'exécution se classent parmi les plus élevés.

Cette étude soulève donc une question importante : doit-on privilégier la rapidité d'un programme au détriment de son optimalité ou au contraire favoriser un résultat minimal ?

En outre, nous pouvons remarquer que lorsque le coefficient de normalisation vaut 1, le temps moyen d'exécution ainsi que le nombre moyen d'états créés est plus faible que pour un coefficient de normalisation valant 4. L'algorithme sous estime le chemin lorsque le coefficient de normalisation est trop élevé.

Comparaison de l'association d'heuristiques Nous allons maintenant nous pencher sur l'utilisation de plusieurs heuristiques afin de résoudre le problème.

	Temps moyen d'exécution en secondes	Nombre moyen d'états créés	Ecart moyen à la solution optimale	Taux d'optimalité
H1 H2 H3 H4 H5 H6	0,077811193	747,6	23	$\frac{1}{10}$
H6 Taux de désordre	0,074782300	1753	0,6	$\frac{7}{10}$
H5 Taux de désordre	0,043249559	913,2	1,8	$\frac{1}{2}$
H2/H4	0,032602048	656,8	12,4	$\frac{1}{5}$

Tout d'abord, nous pouvons noter que lors de l'utilisation simultanée de toutes les heuristiques, l'écart moyen à la solution est très élevé et le taux d'optimalité extrêmement faible. De plus le temps d'exécution moyen se classe parmi les plus élevés.

En outre, deux heuristiques qui nous semblaient intéressantes à combiner étaient la distance de Mahnattan avec une pondération de 1 ainsi que le taux de désordre. En effet, ces deux heuristiques optimales 10 fois sur 10 tentatives mais possédant un temps moyen et un nombre d'espace moyen utilisé très élevé séparément se révèlent être des alliées redoutables lorsqu'elles sont mises en commun. Effectivement, leur écart moyen à la solution optimal est minime : 0,6 et leur taux d'optimalité acceptable : $\frac{7}{10}$.

De même, les heuristiques H5 et le taux de désordre combinés proposent de bonne performances au niveau du temps d'exécution et du nombre d'états moyen créés. Cependant l'écart moyen à la solution optimale est 3 fois supérieur à celui de H6 et Taux de désordre combinés. Le taux d'optimalité est par ailleurs faible : 1 fois sur 2.

Pour finir, nous avons tenté de combiner H2 ainsi que H4, les deux heuristiques qui dissociées présentent les temps moyen ainsi que le nombre moyen d'états créés le plus faible. Toutefois nous pouvons constater que mises ensemble ces dernières présentent des performances inférieures et un taux d'optimalité équivalent.

Chapitre 3

Extensions

3.1 TAQUINS 4*4

Nous allons maintenant nous intéresser à la résolution de taquins 4*4 par notre algorithme. Pour cela nous allons collecter des données relatives au temps moyen d'exécution, le nombre d'états moyens créés et le nombre de coup moyen pour parvenir à résoudre le problème.

De même que lors de la partie précédente, nous collecterons ces données à partir de taquins solubles générés aléatoirement. Ces taquins sont les suivants :

- [0, 7, 3, 12, 14, 13, 5, 9, 10, 6, 11, 15, 1, 2, 8, 4]
- [5, 14, 3, 9, 2, 0, 10, 1, 4, 15, 11, 6, 13, 7, 8, 12]
- [1, 12, 8, 7, 2, 10, 3, 9, 0, 6, 5, 14, 4, 11, 13, 15]
- [13, 11, 12, 8, 15, 1, 0, 3, 7, 14, 9, 4, 5, 2, 6, 10]
- [9, 2, 4, 14, 3, 11, 8, 15, 12, 5, 1, 7, 13, 0, 6, 10]
- [14, 7, 8, 11, 10, 6, 1, 12, 15, 0, 5, 3, 9, 2, 13, 4]
- [4, 7, 5, 0, 13, 11, 1, 12, 15, 10, 9, 3, 8, 2, 14, 6]
- [14, 12, 11, 6, 9, 15, 3, 1, 2, 5, 10, 13, 8, 4, 0, 7]
- [2, 3, 14, 8, 0, 9, 13, 6, 10, 4, 7, 1, 15, 12, 5, 11]
- [7, 15, 2, 12, 1, 6, 5, 3, 10, 4, 11, 14, 13, 0, 8, 9]

Cependant, pour la résolution des taquins 4*4 nous avons pris la liberté de modifier le coefficient de normalisation de H6. Ce dernier vaut désormais $\frac{1}{2}$. Sans cette modification, H6 sous estimait grandement le chemin et par conséquent une exécution du programme avec cette dernière pouvait prendre plus d'une heure.

	Temps Moyen	Etats créés en moyenne	Nombre de coups moyens
H.1	86,852147675	1239867,5	64,1
H.2	2,358877850	34478,3	120,3
H.3	7,183158445	124681,6	67,3
H.4	2,345959473	36172,6	116,9
H.5	2,548957515	45226,2	65,3
H.6	4,902488279	86994,7	61,3
H1/H2/H3/H4/H5/H6	5,448457384	29989,8	159,5
H6/Taux de désordre	4,026388192	50552,5	68,1

Utilisation des heuristiques en simultan  Tout d'abord, nous pouvons observer que lors de l'utilisation simultan e de toutes les heuristiques, l'algorithme sur-estime grandement le chemin et par cons quent le nombre de coup moyen est tr s  lev .

Comparaison des heuristiques s par ment En outre, nous pouvons  galement observer que pour la r solution d'un taquin 4*4, l'heuristique H6 (avec un coefficient de normalisation modifi ) para t la plus judicieuse   utiliser. En effet, elle pr sente le nombre de coup moyen le plus faible et un temps moyen ainsi qu'une utilisation de l'espace m moire raisonnable. Par opposition, l'heuristique H1 semble  tre    viter puisque son temps moyen et son espace m moire utilis  est d raisonnable.

H6/Taux de d sordre Nous remarquons  galement que l'association H6/Taux de d sordre produit un r sultat convenable  galement. Cependant, son nombre de coup moyen  tant sup rieur   H6 utilis e seule, nous ne retiendrons pas cette combinaison.

Conclusion Par cons quent, nous pouvons en conclure que la meilleure heuristique pour r soudre un taquin 4*4 est H6 avec un coefficient de normalisation valant $\frac{1}{2}$.

3.2 TAQUINS 5*5

3.3 Impl mentation d'un autre algorithme de recherche

L'algorithme IDA* Nous avons choisi d'impl menter une variante de l'algorithme A* : IDA*. Son fonctionnement est semblable   celui de A* mais avec un approfondissement it ratif.

L'algorithme fonctionne comme suit :   chaque it ration, on effectue une recherche en profondeur d'abord, en coupant une branche lorsque son co t total $f(n) = g(n) + h(n)$ d passe un seuil donn . Ce seuil commence   l'estimation du co t   l' tat initial et augmente   chaque it ration de l'algorithme.   chaque it ration, le seuil utilis  pour la prochaine it ration est le co t minimum de toutes les valeurs d passant le seuil actuel.

Comme dans A*, l'heuristique doit avoir des propri t s particuli res pour garantir l'optimalit  de la solution. Nous utiliserons donc les m me heuristiques que pr c demment.

3.4 Création de nouvelles heuristiques

- Taux de désordre - H9 (heuristique aléatoire)

3.5 Interface graphique

3.6 Interface Web