Le Taquin

Mélanie MARQUES & Guillaume COQUARD

Rendu le 18 mars 2019

Première partie

Etude Théorique

0.1 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.

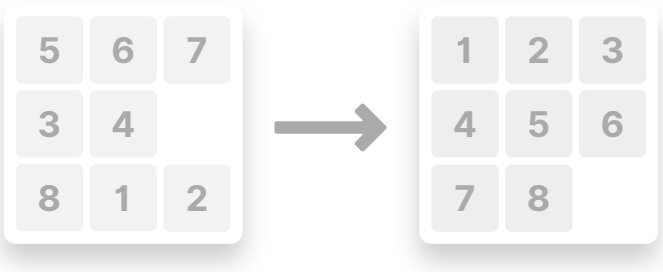


Figure 1 – Un départ possible et l'arrivée voulue

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 ?

0.2 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.

0.2.1 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, réprésenté 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 côut 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

0.2.2 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

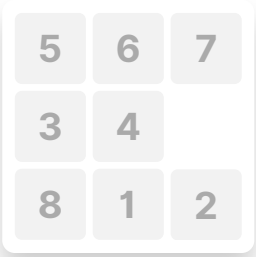
Figure 2 – Tableau des pondérations des distances de Manhattan

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

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

$\varepsilon_E(i)$ 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 cœfficients 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`

0.2.3 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 tuiles 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 accélerer le parcours des états. Ainsi on peut en tirer les conclusions suivantes :

- Si la tuile vide se trouve à l'interieur 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 chacunes des valeurs précédentes.

0.2.4 Définition de l'agent

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

0.2.5 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'éxécution. 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'arreterait 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.

0.2.6 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, rapi-

dement 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, chercher 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 parait 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.

Deuxième partie

Etude Expérimentale

TAQUINS 3*3

Dans cette première partie nous allons nous intéresser aux performances de l'algorithme. Pour cela nous exécuterons le programme sur les taquins suivants (solvables 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.

	Temps moyen d'exécution en secondes	**Nombre moyen d'états créés**	**Ecart moyen à la solution optimale**	**Taux d'optimalité**
H1	0,026284623	623,7	5,2	$\frac{1}{5}$
H2	0,017823386	450,3	9,2	$\frac{1}{5}$
H3	0,049105382	1524	0,6	$\frac{2}{3}$
H4	0,014410663	410,7	8,8	$\frac{1}{5}$
H5	0,042847085	1341,7	0,6	$\frac{2}{3}$
H6	0,087112284	2830,6	0	1
Taux de désordre	1,199918461	36088,5	0	1

Nous remarquons donc que H4 présente le temps d'exécution moyen, et un nombre d'état créé le plus faible cependant elle ne renvoie un résultat optimal qu'une fois sur 5 exécutions.

Au contraire, les heuristiques H6 et Taux de désordre renvoient à chaque exécution un résultat optimal mais leurs temps d'exécution se classent parmis 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 primer 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.

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	$\frac{2}{10}$
H4/H6	0,025508618	523,1	10,2	$\frac{2}{10}$

2 h les plus rapides : H2 et H4 1h rapide 1 h opti