Le jeu du taquin

Jessica DA ROSA – Yannick HONORE

19 mars 2019

Table des matières

1 Introduction

Le taquin est un jeu solitaire en forme de damier crée dans les années 1870 aux Etats-Unis. À partir de 1891, le jeu devient de plus en plus populaire aux Etats-Unis comme en Europe. À l'origine, il est composé de 15 petits carreaux numérotés de 1 à 15 dans un cadre prévu pour 16 carreaux. Le but du jeu est de remettre les carreaux dans l'ordre à partir d'une configuration initiale. Ce principe est retrouvé dans plusieurs autres jeux comme le Rubik's Cube, qui est considéré aujourd'hui comme un descendant du taquin.

2 Projet : Jeu du Taquin

Dans un premier temps, nous allons présenter brièvement en quoi consiste le projet, puis les fonctionnalités qu'il présente et enfin les problèmes qui ont été rencontrés lors de la réalisation de celui-ci ainsi que les solutions qui ont permis de les résoudre.

2.1 Présentation

1. Le jeu du taquin consiste à déplacer le trou dans les directions prédéfinies (nord, surd, ouest, est) à partir d'un état initial jusqu'à un état final en limitant le nombre de déplacements du trou. Pour réaliser cela, nous aurons besoin d'un algorithme écrit avec le langage de programmation Python, de l'algorithme A*, qui a été étudié en cours, et des heuristiques prédéfinies et de la distance de Manhattan servant à calculer les distances élémentaires d'un élément d'un état.

2.2 Fonctionnalités

- L'algorithme se décompose en 2 programmes codés en « Python » :
  - state.py
  - search.py
- Le programme state.py permet de :
  - Créer un taquin avec chaque case à leur place : __init__(self, size).
  - Définir si un mouvement est possible : possible(self, move).
  - Effectuer un mouvement avec le trou dans n'importe quelle direction (nord, sud, ouest, est) : mU(self), mD(self), mL(self), mR(self), mouve(self, toDo).
  - Mélanger le taquin initialisé : shuffle(self, number) et utilisation de la bibliothèque random.
  - Obtenir la distance de Manhatan d'un élément de l'état avec la méthode : distance(self, character).
  - Savoir si un carreau est à la place qu'il doit occuper à l'état final : positions(self).
  - Connaître le nombre de pièces mal placées : nbPieces(self).
  - Pouvoir anticiper quels déplacements je peux effectuer à partir de n'importe quel état : possibilities(self).
  - Vérifier si deux taquins sont identiques : __eq__(self, value).
- Dans la classe Node de :
  - ◦ Définir un état père, un état fils, la fonction d'évaluation f = g + h :__init__(self, stateFather, stateSon, distance, movment).
  - Anticiper les mouvements d'un état fils et calculer la fonction d'évaluation qui en résulte : expansePieces(self). Ici, on prend en compte que l'heuristique est basé sur le nombre de pièces mal placées. ◦ Anticiper les mouvements d'un état fils et calculer la fonction d'évaluation qui en résulte : expanseH(self, k). Pour cette méthode, on prend en compte qu'on expanse en utilisant la distance de manhattan comme heuristique.
  - Réaliser le tableau des heuristiques fourni dans l'énoncé du projet : heuristique(self, k).
- Le programme s1.py permet de :
  - Déterminer si un élémént est présent dans une liste comme l'ensemble frontière ou bien l'ensemble exploré : exist(list, element).
  - Trier l'ensemble frontière lorsque l'on veut y ajouter l'état d'un taquin : TrieInsert(list, element).
  - Rechercher quel est le chemin optimal pour résoudre le taquin : search(start, h).

2.3 Problèmes

Durant la réalisation du projet, nous avons rencontré plusieurs difficultés concernant le mélange aléatoire du taquin, le calcul de et l'affichage du chemin et des taquins de ce chemin.

Au début, nous avions pour idée de créer des tableaux à double-dimentions avec des boucles for pour représenter des taquins. Par ailleurs, il était plus simple d'utiliser la bibliothèque numpy pour créer et manipuler les taquins.

Dans un premier temps, nous avons remarqué que la création d'un taquin directement mélangé aléatoirement était assez complexe à réaliser car nous arrivions sur des résultats présentant des erreurs. Nous avons alors décidé de créer le taquin rempli dans l'ordre croissant pour ensuite pouvoir le mélanger en faisant des mouvements choisit aléatoirement par l'algorithme.

De plus, nous avons mis un temps conséquent à comprendre comment utiliser le tableau des heuristiques fourni dans l'énoncé. Une fois la notion comprise, nous avons décidé de créer un tableau pour y inscrire les coefficients de normalisation afin de pouvoir utiliser un modulo 2 pour nous simplifer l'utilisation de ces coefficients lors du calcul de la formule de h.

Concernant le tableau des heuristiques, il était utilisable pour les taquins 3x3, contenant 8 numéros. Néanmoins, pour les taquins contenant plus de 8 numéros, comme les taquins 4x4 par exemple, nous avons décidé de définir que Hk serait égal à H6 (distance de manhattan) pour obtenir un résultat satisfaisant.

L'affichage du chemin optimal fut un problème assez long mais simple à résoudre. Le premier mouvement ne s'affichait pas car notre boucle while nous permet de remonter de l'état final vers l'état initil en affichant le taquin et la direction utilisée en prenant en compte que pour que l'affichage fonctionne il fallait que l'état actuel ait un état père. Or, l'état initial ne possède pas de père. Il nous a donc falluop rajouté un affichage pour cette exception de la boucle.

3 Travaux d'experiementation

3.1 Melange du taquin et solution sans restrictions

Pour trouver un taquin exploitable on a utilise la fonction shuffle(n) existant dans la class Node. En sachant que n est le numero de mouvements aleatoires que va faire le taquin pour se melanger on a remarquer que plus grand etait n "mieux" mellage etait le taquin ce qui veut dire plus longue est la solution. Des mouvements tires au hasard nous donne pas toujours des mouvements optimaux (Ex : si on a le taquin $\begin{bmatrix} 1 & 3 & 5 \\ 2 & & 7 \\ 6 & 4 & 8 \end{bmatrix}$ on peut faire droite gauche plusieurs fois de suite) ce qui nous rammene a une preiere conclusion :

Notre algorithme n'est pas optimal.

C'est a dire : Si jamais on tombe sur un taquin qui a une taille plus grand que 10 (plus ou moin on n'a pas vraimment trouve un valor exact) on ne va pas toujours trouvee une solution. Ce qui nous impeche pas de continuer a l'explorer.

3.2 Heuristiques

Avec l'implementation des heuristiques avec la formule : $h_k(E) = (\sum_{i=0}^{8} \pi_i \times \epsilon_E(i)) div \rho_k$ la variation des resultats comence a augenter :

- si o a un taquin avec une solution optimal avec moins de 10 mouvements de la case vide on va une soluiton presque intatanement avec n'importe quel heurstique (les variations dans le temps d'execution sont de quelques secondes ou meme moins)
- mais si on se trouve face a face avec une solution un peut plus longue on a des chances que chaque recheche de solution tombe sur un valeur differant voir meme pas de resultat du tout (dans ce cas on constate une vrai faille dans l'ago concu)

Exemple :

On mantient le tableau :

	1	2	3	4	5	6	7	8	0
π_0	36	12	12	4	1	1	4	1	0
$\pi_1 = \pi_2$	8	7	6	5	4	3	2	1	0
$\pi_3 = \pi_4$	8	7	6	5	3	2	4	1	0
π_5	1	1	1	1	1	1	1	1	0

$\Rightarrow$Distance de Manhatan

Si on prend come depart le taquin : $\begin{bmatrix} 2 & 8 & 4 \\ 1 & 6 & 3 \\ 7 & 5 & \end{bmatrix}$

On pet trouver les valeurs suivants por l'execution :

	ρ_k	temps(s)	temp (min)	solution	len(sol.)
h_0	4	261,6737	4,3612	['L', 'U', 'U', 'R', 'D', 'L', 'U', 'L', 'D', 'R', 'R', 'D', 'L', 'U', 'R', 'D']	16
h_1	1	undetermine		introuvee	
h_2	4	1,4319	0,0238	['L', 'U', 'U', 'R', 'D', 'D', 'L', 'U', 'U', 'L', 'D', 'R', 'R', 'D']	14
h_3	1	undetermine		introuvee	
h_4	4	1,0895	0,0181	['L', 'U', 'U', 'R', 'D', 'D', 'L', 'U', 'U', 'L', 'D', 'R', 'R', 'D']	14
h_5	1	0,5783	0,0096	['L', 'U', 'U', 'R', 'D', 'D', 'L', 'U', 'U', 'L', 'D', 'R', 'R', 'D']	14

Tout suite on remarque que pour h_1 et h_3 on ne trouve pas de solution. Ce fenomene est produit du a la saturation de la memoire de l'a machine que au bout de 2h n'avais plus despace pur stocker les etats a traiter.

Pour tant pour un taquin un peut plus simple comme par example : $\begin{bmatrix} 1 & 6 & 2 \\ 7 & & 3 \\ 5 & 4 & 8 \end{bmatrix}$

On pet trouver les valeurs suivants por l'execution :

	ρ_k	temp(s)	temp (min)	solution	len(solution)
h_0	4	0,0504	0,0008	['U', 'R', 'D', 'L', 'D', 'L', 'U', 'R', 'D', 'R']	10
h_1	1	0,0159	0,0003	['D', 'L', 'U', 'R', 'U', 'R', 'D', 'L', 'D', 'R']	10
h_2	4	0,1088	0,0018	['D', 'L', 'U', 'R', 'U', 'R', 'D', 'L', 'D', 'R']	10
h_3	1	0,0154	0,0003	['D', 'L', 'U', 'R', 'U', 'R', 'D', 'L', 'D', 'R']	10
h_4	4	0,0906	0,0015	['D', 'L', 'U', 'R', 'U', 'R', 'D', 'L', 'D', 'R']	10
h_5	1	0,0369	0,0006	['D', 'L', 'U', 'R', 'U', 'R', 'D', 'L', 'D', 'R']	10

On peut remarquer que :$(h_3 = h_1) < h_5 < h_0 < h_4 < h_2$ par apport au temps de resolution du taquin.

Le resultat de ces deux experiences est congruent avec la conlusion faite au debut sur l'efficacite de l'algorithme dans son etat actuel vu que d'un cote on ne trouve pas de solution valable et de l'autre on a une solution en temps optimal.

Les resultat theoriques trouves peuvent etre apliques a des taille plus grades de taquin en utilizant le meme algorithme et en changeant juste quelques element dans le fichier de texte. Pour pouvoir obtenir des resultats pour d'autres valeurs de π_k on a finit par inserer une clause dans la determination de $h_k$qui pour des $E > 8$ va nous donner la distance de manhatan tout simple (un rajout de valeurs au tableau aurait etait peutetre preferale en donnat des valeures experimentales plus interessants vu on se retouve havec beaucoup de f qui ont la meme valeur ce qui ne facilite pas la recherche)

Exemple

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	0
π_0	36	12	12	4	1	1	4	1	1	1	1	1	1	1	1	0
$\pi_1 = \pi_2$	8	7	6	5	4	3	2	1	1	1	1	1	1	1	1	0
$\pi_3 = \pi_4$	8	7	6	5	3	2	4	1	1	1	1	1	1	1	1	0
π_5	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0
$\pi_6 = \pi_7$	36	35	34	33	12	12	16	4	8	1	1	4	1	2	7	0

on va experimenter sur le taquin : $\begin{bmatrix} 6 & 1 & 3 & 4 \\ & 5 & 7 & 8 \\ 13 & 2 & 10 & 11 \\ 14 & 9 & 15 & 12 \end{bmatrix}$

	ρ_k	**temp(s)**	**solution**	**len(sol)**
h_0	**4**		**introuvee**	
h_1	**1**	**133.92**	**['R','D','L','U','U','R','D','L','D','R','D','L','U','R','R','R','D']**	**17**
h_2	**4**		**introuvee**	
h_3	**1**	**54.4177**	**['R','D','L','U','U','R','D','L','D','R','D','L','U','R','R','R','D']**	**17**
h_4	**4**		**introuvee**	
h_5	**1**		**introuvee**	
h_6	**4**	**19.1654**	**['R','D','L','U','U','R','D','L','D','R','D','L','U','R','R','L','R','R','D']**	**19**
h_7	**1**		**introuvee**	

On remarque, a traver des differents temps d'execution obtenues que les valeurs de $h_k$dans la formule aident a que le calcul soit effectue plus rappidement. on pet voir le effet de ces valeurs dans la difference de temps entre h_0 et h_6 ou pour les cases avec des valeurs de 1 a 8 (compris) on ha des $\pi_i[k]$ plus grans que 1 et les restante elages a un et pour l'autre la majorite des casesont des valeurs superieurs a un, pour le premie on n'a pas de resulat(la memoire est epuise avant qu'on latteint) et pour l'autre on a un resultat en 19s plus ou moins.

ca nous ammene a la conclusion que un sisteme de $\pi_i[k]$ existe (meme si o ne l'a pas trouve) et que ce sisteme aide a la elimination eficace de cas redeondants.

La valeur qu'on remarque le plus c'est ρ_k que quand a 1 ha tendance a prelonger la recherche jusqu'au epuisement de la memoire (ce fenomene est plus evidant dans le taquin 4x4 represente en haut) et qund a 4 fait dinminuir le temps de calcul. un $\rho_k = 5$ diminue considerablement le tempd'execution on arrive a trouver une solution en 30s avec les ensemples de $\pi_i[k]$ utilisees, Manhatan retourne une solution enn 50s.

Apres multiples lancements de la recherche avec le meme taquin initial et le meme valeur de k pourla determination de l'heuristique utilise on remarque que les valeurs de g et h evouent de facon inverse. On verifie que h_k a tendance a diminuer et devenir 0 pour les derniers etats de la solution ce qui donne a f la valeur de g uniquement vue que $f = g + h$.

3.3 Conclusion

Meme si avec un algorithme qui n'est pas tout en effet optimal pour la echerche on peut conclure que :

1. un $\rho_k! = 1$ est important pour acceler unre recherche quand on a une longue solution
2. h et g evoluent en oposition l'un de l'autre tant que g augmente avec la profondeur de l'arbre de recherche h lui a tendance a dimminuer ce qui nous donne un valeur de f = g pour les neuds les plus profonds vu que $f = g + h$.
3. il existe un sensemble $\pi_i[k]$ optimal pour la resolution d'un taquin et ce sisteme va etre adapte a la taille du taquin a resoudre. la valeur la plus grande sera celle atribue a la case contenat 1 et la casevide(celle qui contient 0 dans notre representation) va avoir 0 come valeur