Rapport de projet

Résolution de Taquin - Intelligence Artificielle

Professeur responsable : Jean-Marc Salotti
Equipe projet : Hugo Fournier et Benjamin Nicol

I - Le jeu de Taquin

A. Présentation

Le Taquin est un jeu solitaire créé aux Etats-Unis dans les années 1870. Ce jeu est composé d'un damier d'une taille de 4 de largeur pour 4 de longueur, avec donc 16 cases remplies de 15 tuiles coulissantes. Elles sont numérotées de 1 à 15, le but de ce jeu est de mettre dans l'ordre les tuiles dans le sens de lecture (de droite à gauche puis de haut en bas) en faisant coulisser les cases en utilisant l'emplacement laissé vide.

Lors de ce projet nous allons utiliser des variantes du jeu de taquin. Nous travaillons sur un premier taquin taille 3x3 comportant deux trous et un deuxième de taille 5x5 comportant lui aussi 2 trous.

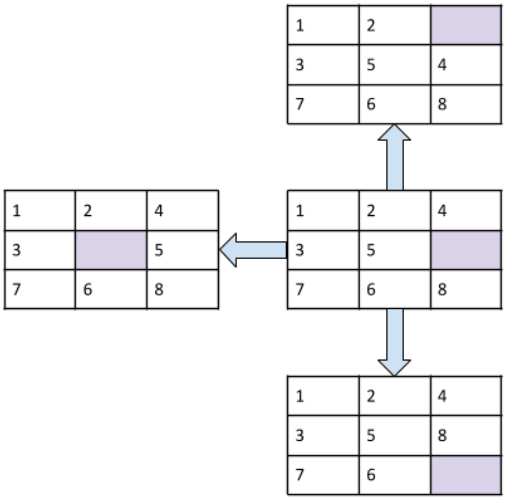
B. Techniques d'IA

Le jeu de taquin, de part son format, est un jeu très souvent utilisé en intelligence artificielle pour montrer l'utilisation des algorithmes de parcours de graphes. Ce jeu sert souvent d'introduction à l'algorithme de Dijkstra et A*, ainsi qu'à l'utilisation d'heuristiques étant donné que certaines configuration d'un jeu de taquin de taille 3x3 ne sont pas résolvables sans utilisation d'heuristique.

II - Modélisation et algorithmes

A. Modélisation du problème

Pour résoudre le problème du taquin, il faut d'abord le modéliser. La résolution d'un taquin de taille n*n peut se modéliser par une succession d'états représentant les coups nécessaires pour arriver à l'état final. Ainsi, voici une représentation d'un état d'un taquin 3*3 à 1 trou et ses successeurs possibles :



Une notion primordiale est celle de l'état final, cela représente le plateau de taquin résolu, lorsque tous les chiffres sont à la bonne place, comme pour ce taquin 5*5 à deux trous par exemple :

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23		

Ainsi, la résolution d'un taquin peut se modéliser par le parcours d'un graphe acyclique, connexe (il est toujours possible d'aller d'un sommet à un autre en empruntant les arêtes) et orienté car il faut pouvoir tenir compte du nombre de coups et de l'avancement de la résolution. Chaque coup coûtant un tour, le coût d'une arête allant d'un taquin à son successeur est toujours de 1 (pas de coûts différents comme sur une modélisation de réseau routier par exemple).

B. Algorithmes

a. Algorithme A étoile

Pour expliquer A* il est tout d'abord nécessaire d'aborder l'algorithme de Dijkstra. Il est utilisé pour trouver le plus court chemin entre deux noeuds dans un graphe. Cet algorithme va utiliser une stratégie de parcours en profondeur sauf que le placement des noeuds dans la liste des ouverts est défini par une stratégie différente du parcours en profondeur classique. Les premiers dans la liste des ouverts sont les noeuds dont le coût est le plus faible, c'est à dire que le poids de l'arête reliant les noeuds est minimal.

A* est une amélioration de l'algorithme de Dijkstra, il utilise la même base sauf que le coût est calculé non plus uniquement avec le poids de l'arête mais aussi avec une heuristique. Si celle-ci est minorante, cela permet à A* de trouver le plus court chemin entre deux sommets beaucoup plus rapidement qu'avec un simple algorithme de Dijkstra.

b. Algorithme A ttrapay

Il s'agit de l'algorithme A étoile mais quand les coûts heuristiques commencent à être faible, n'importe quel noeud ouvert qui serait terminal est directement choisi.

C. Heuristiques

Pour les heuristiques de l'algorithme A étoile, nous nous sommes intéressés aux 2 heuristiques les plus connues dans la littérature scientifique, réputées pour bien fonctionner.

a. Heuristique W

L'heuristique W ou "Nombre de cases mal remplies" est égale au nombre de cases du plateau sur lesquelles il y a le mauvais nombre. Comme il faudra au moins bouger une fois chacune de ces cases, on voit facilement que W est minorant du coût total du chemin jusqu'à l'état final ; ceci nous assure donc que l'utiliser avec A* donnera toujours la solution avec le nombre minimal de coups.

Pour mieux illustrer cette heuristique, voici un exemple dans un taquin à un trou :

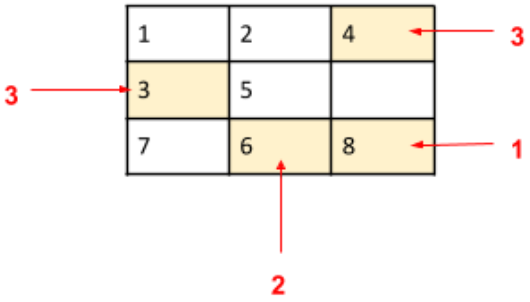
1	2	4
3	5	
7	6	8

Ici W vaut donc 4.

b. Heuristique P

L'heuristique P ou norme Manhattan est égale à la somme des distances de chaque nombre à sa case finale (en nombre de coups). Comme il faudra au minimum bouger chaque case en un nombre de coups égal à cette distance, P est aussi un minorant du coût total du chemin jusqu'à l'état final; on peut donc l'utiliser avec A* en garantissant une solution avec le nombre minimal de coups.

Pour mieux comprendre cette heuristique, reprenons le même exemple que tout à l'heure, avec la participation à P pour chaque nombre mal placé fléchée en rouge :



Ici, P vaut donc 3+3+1+2 = 9.

c. Heuristique P175

L'heuristique P175 est une heuristique que nous avons testée, avec succès, pour résoudre les taquins 5*5. Il s'agit simplement de l'heuristique P à la puissance 1.75. Ainsi, ce n'est plus un minorant, et donc elle ne garantit pas une solution en nombre minimal de coups avec A* . Mais elle permet d'arriver à une solution bien plus vite en priorisant plus fortement dans la liste des noeuds ouverts les taquins proches de l'état final.

III. Implémentation

Dans cette partie, nous expliquerons nos choix d'implémentation qui diffèrent du code source proposé par nos encadrants en tant que base.

NodeTaquin

NodeTaquin hérite de la classe abstraite *GenericNode* et redéfinit ses méthodes pour pouvoir être utilisé dans les algorithmes de *SearchTree*.

Pour faciliter les différents calculs (notamment les heuristiques) nous avons codé l'état d'un noeud taquin par l'attribut *state*, qui est un tableau à deux dimensions d'entiers. Les nombres correspondent à la valeur de la case, et -1 est un trou.

Les variables *NbHoles* et *SizeTaquin*, construites en analysant le plateau donné au constructeur, sont utiles pour plusieurs méthodes, et permettent à cette classe, *NodeTaquin*, de fonctionner pour n'importe quelle taille de taquin (tant que le taquin est bien carré).

Pour l'implémentation des heuristiques, dans la méthode *CalculeHCost* il suffit de décommenter le bloc correspondant et commenter l'autre pour passer de l'heuristique P à l'heuristique W.

Pour pouvoir créer les taquins successeurs dans *GetListSucc()* nous avons implémenté la méthode *switch2numbers(i1,j1,i2,j2)* qui renvoie un tableau d'état avec deux nombres inversés, ainsi que la méthode *shallowCopy (source, sizeTaquin)* qui permet d'obtenir une copie superficielle et donc d'éviter les effets de bord malvenus.

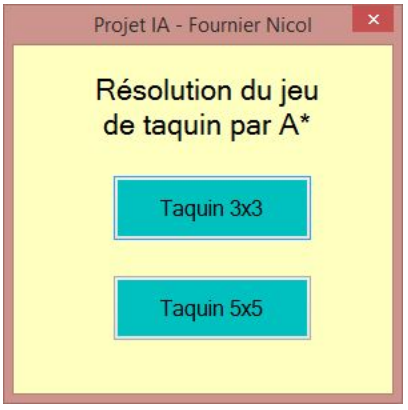
Enfin, pour faciliter l'affichage dans l'interface, nous avons implémenté un *ToString()* correspondant à la modélisation en string des états du taquin proposée dans le code de base.

A. SearchTree

Ici, comme les algorithmes fonctionnaient déjà bien, la seule différence est l'implémentation de l'algorithme A ttrapay décrit ci-dessus.

B. Interface

Le design de l'interface a été simplifié au maximum afin de passer plus de temps sur le code que sur le design. Elle est constituée d'une page d'accueil permettant de choisir une taille de taquin, une fois le bouton voulu cliqué, cela ouvre une nouvelle fenêtre avec le taquin de la taille voulue.



Fenêtre de sélection de taille de taquin

Fenêtre pour un taquin 3x3

L'interface de résolution d'un taquin est divisé en 4 zones :

- La zone **Taquin** : qui contient 9 bouton en lecture seule pour afficher le taquin (les trous seront indiqués par des '-1')
- La zone **Résultats** : qui contient le temps de calcul pour résoudre le taquin, ainsi que le nombre de noeuds ouverts et fermés
- La zone **Actions** : contenant le choix du taquin, représenté par trois boutons A, B et C permettant de sélectionner les taquins à résoudre présentés dans le sujet de ce projet; ainsi que le bouton résoudre qui va appeler l'algorithme de résolution du taquin et ensuite afficher les résultats
- Et enfin une dernière zone regroupant la liste des coups pour gagner et l'arbre d'exploration qu'a réalisé l'algorithme afin de résoudre le taquin sélectionné

Notre interface ne dispose que de très peu de code. En effet, elle se sert des autres classes du programme afin de fonctionner. Les algorithmes de résolution sont ceux de la classe SearchTree, et les résultats affichés sont aussi directement récupérés depuis cette classe ; seul un chronomètre à été implémenté afin de connaître le temps d'exécution de la recherche.

Il est aussi à noter que les différents taquins ont été écrit en dur dans le code, aucun taquin aléatoire n'est utilisé, seulement les taquins demandés dans les exercices du projet, pour permettre une comparaison dans les tests de performance.

IV - Résultats

A. Exercice 1

a. Sans heuristique

Interfaces de résultats pour une résolution sans heuristique

	Temps de calcul	Noeuds ouverts	Noeuds fermés
Taquin A	00:01.82	2578	2632
Taquin B	00:48.46	10049	15302
Taquin C	Non résolu en plus de 5 minutes	/	/

b. Heuristique W

	Temps de calcul	Noeuds ouverts	Noeuds fermés
Taquin A	00:00.01	264	183
Taquin B	00:00.18	932	746
Taquin C	00:10.31	5928	6359

c. Heuristique P

	Temps de calcul	Noeuds ouverts	Noeuds fermés
Taquin A	00:00.00	67	40
Taquin B	00:00.00	198	135
Taquin C	00:00.04	443	315

d. Comparaison

		Temps de calcul	Noeuds totaux (ouverts + fermés)
Taquin A	Sans heuristique	00:01.82	5210
	Heuristique W	00:00.01	447
	Heuristique P	00:00.00	107
Taquin B	Sans heuristique	00:48.46	25531
	Heuristique W	00:00.18	1678
	Heuristique P	00:00.00	333
Taquin C	Sans heuristique	/	/
	Heuristique W	00:10.31	12287
	Heuristique P	00:00.04	758

Le tableau ci-dessus rassemble les résultats obtenus sur le taquin 3x3 avec 2 trous en fonction des différentes heuristiques.

On peut noter assez rapidement que sans l'utilisation d'heuristique, le problème devient vite assez complexe à résoudre. En effet, on remarque que dès le taquin B, le programme met 48 secondes pour trouver la succession d'actions gagnantes et il n'arrive même pas à résoudre le taquin C en moins de 5 minutes.

On peut voir que l'heuristique W qui correspond aux cases mal placées explore moins de noeuds que sans heuristique et résout le problème bien plus rapidement, 0.18 secondes contre 48.46 secondes pour le taquin B.

Et enfin, l'heuristique P, la distance de Manhattan, nous permet d'obtenir des temps de calculs amoindris comparés aux deux autres techniques mais aussi de réduire considérablement le nombre de noeuds, environ 12 000 pour l'heuristique W contre 758 pour l'heuristique P pour le taquin C.

On observe donc une influence majeure de l'heuristique sur la vitesse de résolution d'un taquin en utilisant l'algorithme A*.

e. Algorithme A ttrapay

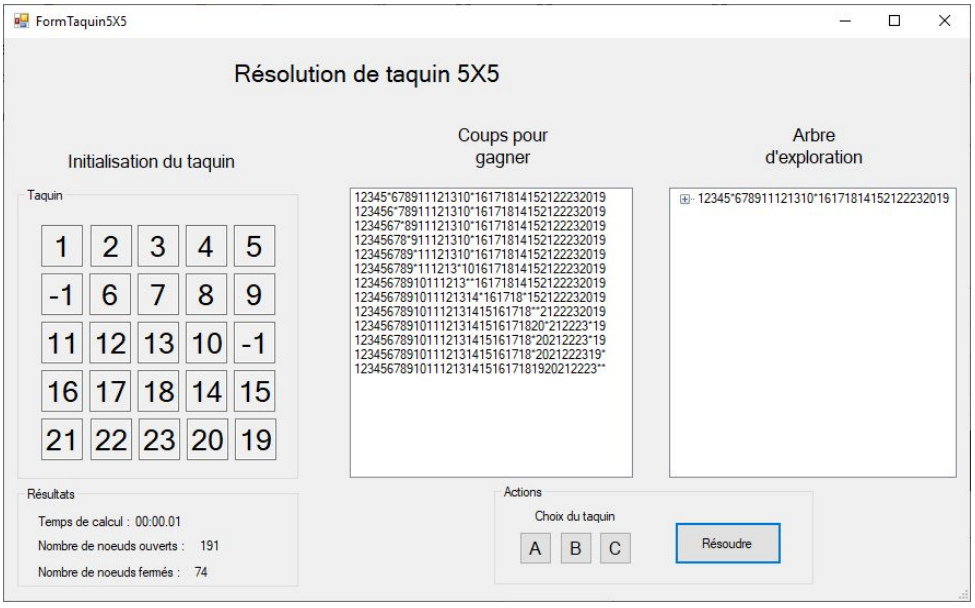
Sur les taquins 3*3, l'algorithme A ttrapay ne réduit que très sensiblement le nombre de noeuds ouverts et le temps de calcul.

B. Exercice 2

a. Heuristique P

Pour résoudre l'exercice deux qui consistait en 3 taquins de taille 5x5 avec 2 trous. Nous avons dû utiliser d'autres techniques que celles vues pour le premier exercice car un taquin 5x5 avec deux trous peut créer des graphes de résolution énormes. L'utilisation de l'heuristique P a été indispensable, étant donné que l'heuristique W ne donnait aucun résultat avec 5 minutes de temps de calcul.

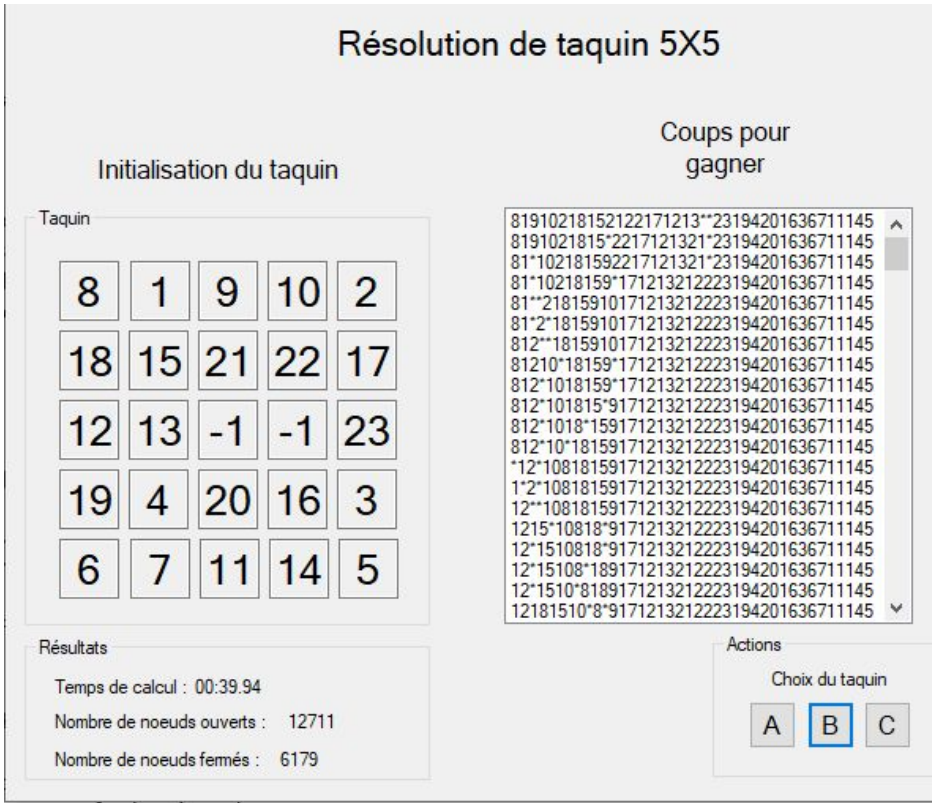
Sur ce premier exemple, on peut voir que le taquin A est assez simple à résoudre et que l'algorithme A trrapay avec une heuristique P classique trouve une succession de coup gagnants très rapidement en peu de noeuds et en peu de coups.

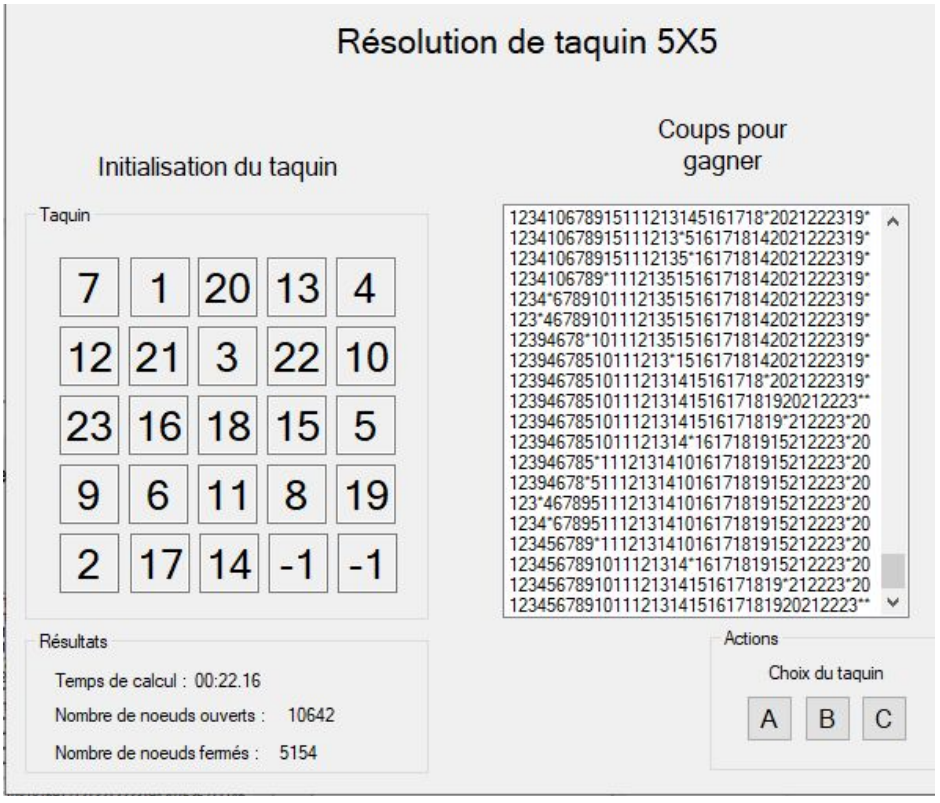


Taquin A résolu avec une heuristique P

Cependant, les taquins B et C n'ont pas été résolus avec cette heuristique en moins de 5 minutes, nous avons donc décidé de modifier légèrement l'heuristique pour accélérer le calcul. Nous avons donc implémenté l'heuristique P175 comme expliquée plus haut.

b. Heuristique P175





Taquin B (à gauche) et C (à droite) résolus à l'aide de l'heuristique P175

Nous pouvons voir ci-dessus les écrans de résultat obtenus à l'aide de l'heuristique P175. Nous constatons que les temps de résolution sont inférieurs à la minute bien que le nombre de noeuds total soit important (entre 15 000 et 20 000).

Comme dit précédemment, cette heuristique permet de trouver une solution mais n'est pas minorante, on observe donc que les listes des coups pour gagner sont assez importantes et ne sont probablement pas celles possédant le moins de coups possible.

V - Répartition des tâches

Pour ce projet nous avons presque exclusivement travaillé en binôme en présentiel, pour favoriser la communication et le suivi des avancées. Voici ce qui a été réalisé à chaque séance et par qui.

Séance	Hugo	Benjamin
15/10/19 (3h)	Implémentation des méthodes de NodeTaquin : IsEqual, GetArcCost, EndState, CalculeHCost	Création des interfaces Winforms : main, taquin 3x3, taquin 5x5
15/11/19 (3h)	Finition de NodeTaquin et intégration du taquin 3x3 avec l'interface	Implémentation des taquins de référence et nouveaux indicateurs de résultats
Entre sessions (< 1h)	Légers fixes pour sortir la version 1 (heuristique W)	
26/11/19 (2h00)	Travail non fini sur l'implémentation de l'heuristique P + ToString() du NodeTaquin	Nouveaux indicateurs de résultats sur la résolution et de l'arbre d'exploitation
3/12/19 (2h20)	Implémentation finie de l'heuristique P	Intégration du 5x5, fix du treeview et du chronomètre
5/12/19 (2h)	Travail non fini sur l'algorithme A attrapay + plan du rapport + rédaction II.A.	Améliorations de l'interface + tests et rapports des résultats + rédaction IV. A.
Entre sessions (> 2h)	Coloration de l'interface + Implémentation et test de l'heuristique P175 + Rédaction partie IV + partie III.A + partie III.B + partie II. A. + partie II. C. + partie V.	Rédaction partie I. + partie II. B. + + partie IV. B.
11/12/19 (1h)	Relecture version 1 du rapport	Relecture version 1 du rapport
Entre sessions	Validation et envoi du rapport	Rédaction partie III. C. + partie I.B.