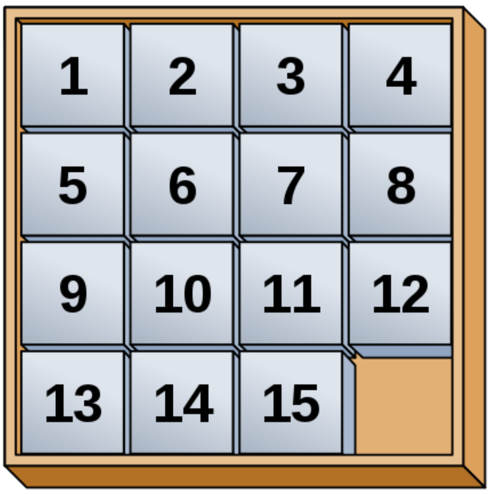
INSTITUT NATIONAL
DES SCIENCES
APPLIQUÉES
ROUEN NORMANDIE

Contraintes et Programmation Logique

Le Jeu du Taquin



Quentin LOISEAU
Rand ASSWAD
Génie Mathématique

A l'attentien de :
M. Habib ABDULRAB

Table des matières

1 Introduction

1.1 Le jeu du taquin

Le jeu du taquin est un puzzle qui a été créé vers 1870 et a attiré l'intérêt de nombreux mathématiciens pour sa valeur en tant qu'un problème combinatoire.

Le jeu est composé de $n \cdot m - 1$ petits carreaux numérotés à partir de 1 qui glissent dans un cadre du format $n \times m$ laissant une case vide permettant de modifier la configuration des carreaux. Le jeu consiste à remettre dans l'ordre ces cases à partir d'une configuration initiale quelconque.

Le jeu souvent connu dans les formats 3×3 ou 4×4, d'où l'appélation anglophone *8-puzzle* ou *15-puzzle* respectivement.

Le Rubik's Cube est condéré comme l'un des descendants du taquin. (Wikipédia 2018)

1.2 Complexité

Le jeu du taquin est le problème le plus grand de son type qui peut être résolu complètement (trouver toutes les solutions du problème à partir d'un puzzle solvable). Il est simplement défini mais le problème est **NP-difficile**. (Reinefeld 1993)

Le problème est grand combinatoirement est exige une résolution guidée afin d'atteindre avant d'épuiser les résources (temps et mémoire).

L'espace d'état est de taille $\frac{(n \cdot m)!}{2}$ ce qui fait 181440 pour la variante 3×3.

En effet, il existe $(n \cdot m)!$ permutations des tuiles, une permutation est solvable, l'argument de parité a été présenté dans (Johnson et Storey 1879) afin de montrer que la moitié des configurations initiales de ne peut jamais pas atteindre l'état but en définissant une fonction invariante de mouvement des tuiles qui définit deux classes d'équivalence d'états *accessibles* et *non-accessibles*.

1.3 Le but du projet

Le but du projet est de résoudre un problème réel à l'aide de la programmation logique. Notre choix s'est portée le jeu du taquin car il est intéressant en tant qu'un problème d'Intelligence Artificielle, et s'adapte bien à la programmation logique.

Dans ce projet on présentera ce problème mathématiquement et proposons des algorithmes de résolutions variées, une implémentation en **prolog** de ces algorithmes, et finalement une étude des résultats.

1.4 Formulation du problème

Le problème est le mieux représenté par un graphe connexe. On défini l'espace d'état E par toutes les configurations solvables du problème.

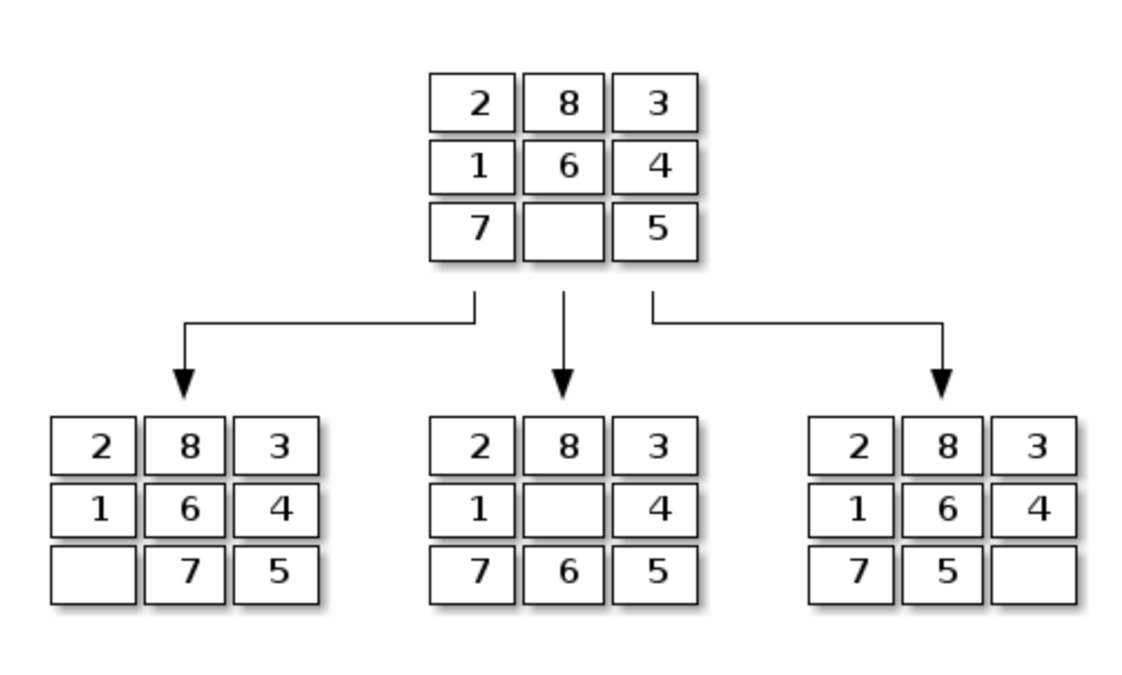


Figure 1 – Une partie du graphe du taquin

On considère deux états adjacents si on peut passer de l'un vers l'autre en glissant une seule tuile voisine vers la case vide. La case vide a au moins 2 tuiles voisines (si elle est dans un coin), et au plus 4 (gauche, droite, haut, bas).

On cherche idéalement le chemin le plus court pour arriver au but à partir de l'état initiale, ceci consiste un problème classique de recherche de chemins dans un graphe.

2 Algorithmes

Ils existent de nombreux algorithmes pour la recherche d'un chemin dans un graphe, nous avons implémentés quelques algorithmes qui correspondent bien à notre problème et qui s'adaptent bien aux principes de la programmation logique.

2.1 Depth-First Search (DFS)

L'algorithme de parcours en profondeur (DFS) est un algorithme complet permettant de trouver un chemin dans un graphe.

Cette algorithme est le principe *inné* de **prolog** de l'arbre de résolution.

L'implémentation de DFS dans un arbre se fait simplement par le code

```
dfs(Etat, [Etat]) :- final(Etat).
dfs(E1, [E1|Chemin]) :-
    adjacent(E1, E2),
    dfs(E2, Chemin).
```

On obtient notre chemin par la requête

```
?- dfs([EtatInitial], Chemin).
```

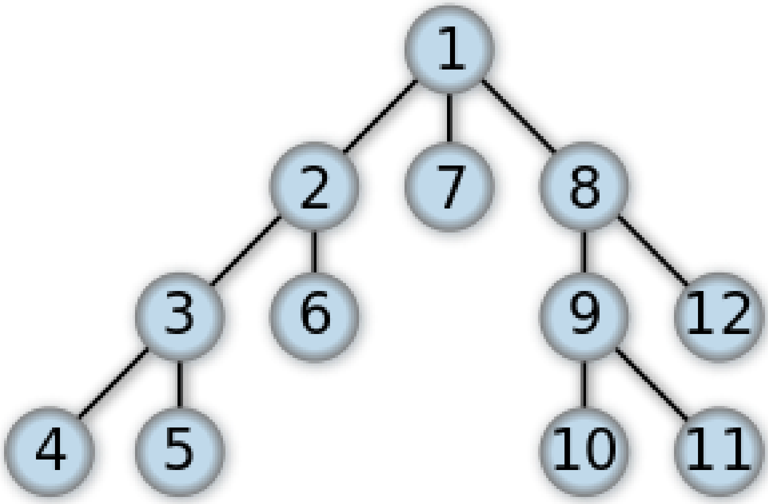


FIGURE 2 – L'ordre de parcours des nœuds dans l'algorithme DFS

Néanmoins, cet algorithme ne fonctionne pas pour la plupart des puzzles; il parcourt en profondeur donc il prendra toujours le premier adjacent de `E1` jusqu'à ce que le dernier `E1` n'a plus d'adjacents, et puis tentera le deuxième adjcent du `E1` précedant, et ainsi suite. . . sauf que dans le jeu du taquin *il y a toujours au moins 2 adjacents!* le programme est donc infiniment récursif.

Si on suppose que le prédicat `adjacent/2` est défini par

```
adjacent(A, B) :- adjacent(A, B, gauche).
adjacent(A, B) :- adjacent(A, B, droite).
adjacent(A, B) :- adjacent(A, B, haut).
adjacent(A, B) :- adjacent(A, B, bas).
```

L'arbre de résolution de prolog dépend de l'ordre de définition des prédicats, il tentera les adjacents dans l'ordre (gauche, droite, haut, bas) donc pour l'état suivant qui est adjacent au but, il modulera entre ces deux états jusqu'à ce qu'il na plus de mémoire.

```
1 2 3      1 2 3      1 2 3      1 2 3      1 2 3
4 5        4 5        4   5      4 5        4   5
7 8 6      7 8 6      7 8 6      7 8 6      7 8 6      ...
```

Il est donc nécessaire d'interdire de prendre un adjacent `E2` déjà visité. Or, un nouveau problème s'introduit:

```
1 2 3      1 2 3      1 2 3      1 2 3      1 2 3
4 5        4 5          4 5      7 4 5      7 4 5
7 8 6      7 8 6      7 8 6        8 6      8   6      ...
```

Nous avons testé cet algorithme avec cette configuration, il fait 27 mouvements afin d'arriver au but ! Théoriquement, il trouvera toujours un chemin, mais en pratique pour la plupart des configurations le overflow arrive avant de trouver une solution.

D'où la nécessité de prendre des décisions informées, nous allons ainsi introduire la notion d'**heuristique**.

2.2 Heuristiques

Une **fonction heuristique** sur un graphe est une fonction $h : E \to \mathbb{N}$ où E est l'espace d'états du problème, $h(n)$ représente le coût estimé pour arriver au but à partir du nœud n.

On appelle **heuristique admissible** une heuristique h telle que

$$\forall n \in E, h(n) \leq h^*(n)$$

où $h^*(n)$ est le vrai coût minimal pour arriver au but à partir du nœud n (dite l'*heuristique parfaite*).

Trivialement, l'heuristique nulle est admissible mais elle ne rajoute aucune valeur à la résolution du graphe. (Wikipedia contributors 2018)

Nous allons introduire deux heuristiques qu'on a utilisé dans nos algorithmes.

2.2.1 Distance de Hamming

La distance de Hamming est définie pour deux listes (ou mots) de même taille par le nombre de valeurs qui diffèrent entre ces deux listes.

Soit mathématiquement, avec A l'ensemble des atoms (ou alphabet)

$$\begin{aligned} d_{\text{Hamming}} &: A^n \times A^n \to \mathbb{N} \\ (x,y) &\mapsto \sum_{i=1}^{n} (1 - \delta_{x[i],y[i]}) = \begin{cases} 0 & \text{si } x[i] = y[i] \\ 1 & \text{sinon} \end{cases} \end{aligned}$$

Dans notre contexte, on ne prend pas en compte de la case vide dans ce calcul.

L'heuristique de Hamming est donc la distance de Hamming entre le tableau du nœud n et le nœud final. Cette heuristique est admissible.

2.2.2 Distance de Manhattan

La distance de Manhattan est la distance L_1 dans les espaces de Banach. Soit V un espace de Banach de dimension n.

$$\begin{aligned} d_1 &: V \times V \to \mathbb{R}_+ \\ (x,y) &\mapsto \sum_{i=1}^{n} |x_i - y_i| \end{aligned}$$

Dans notre contexte, la distance est définie sur $\{1,\ldots,n\} \times \{1,\ldots,m\}$ et ne prend pas en compte de la case vide.

De même, l'heuristique de Manhattan est la distance entre le nœud n et le nœud final. Cette heuristique est admissible.

État n			Final		
8	1	3	1	2	3
4		2	4	5	6
7	6	5	7	8	

Heuristique	Hamming								Manhattan								h^*
Tuile	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8	
Distance	1	1	0	0	1	1	0	1	1	2	0	0	2	2	0	3	
Total	5								10								14

FIGURE 3 – Heuristiques de Hamming et de Manhattan

2.3 Algorithme Greedy

Dans le cas du jeu du taquin, l'algorithme de Greedy est un algorithme d'optimisation locale; lorsqu'il faut faire un choix parmi une liste d'adjacents il prend l'adjacent qui minimise la fonction coût.

L'algorithme est complet mais pas toujours optimale, dans notre implémentation nous avons obtenue les meilleurs résultats pour l'heuristique définie par

$$h(n) := \text{Manhattan}(n) + 3 \cdot \text{Hamming}(n)$$

Cette heuristique n'est pas admissible car elle vaut pour un état n adjacent au but

$$h(n) = (1) + 3(1) = 4 > 1 = h^*(n)$$

mais cela n'a aucune importance dans cet algorithme.

L'algorithme trouve une solution en quelques secondes pour toutes les configurations du taquin de taille 3×3, la plupart des solutions ne sont pas optimales.

Pour les tailles plus grandes, l'algorithme prends plus de temps et ne trouve pas toujours une solution (overflow ou timeout).

2.4 Iterative Deepening DFS (ID-DFS)

L'algorithme ID-DFS est une variante du DFS, il effectue une recherche en profondeur DFS itérativement pour un profondeur limite donnée d, et l'incrémente succéssivement en commençant par $d = 0$ jusqu'à ce qu'il trouve une solution. (Wikipedia contributors 2019b)

Classiquement, $d = 0$ à la première itération mais cela est loin d'être optimale, nous proposons donc de commencer par une sous-estimation du longeur du chemin, $d = h(n_{\text{initial}})$ est une sous-estimation si h est une heuristique admissible.

Compléxité

- Compléxité en temps: $O(b^d)$
- Compléxité spaciale: $O(d)$

où d est le profondeur, et b est le facteur de branchement.

L'algorithme trouve en quelques secondes des solutions optimales pour toutes les configurations de taille 3×3, et trouve rarement des solution pour les tailles plus grandes.

Ceci est dû au fait qu'une fois d est grand, ID-DFS est presque comme DFS et a les mêmes problèmes.

2.5 Algorithme A*

L'algorithme A* est un complet, optimal et efficace. C'est un algorithme *à mémoire*, qui a une compléxité spaciale importante.

A* utilise la fonction d'évaluation f défini par

$$f(n) = g(n) + h(n)$$

où

- $f(n)$ le coût total estimé au nœud n
- $g(n)$ le vrai coût pour arriver au nœud n à partir du nœud initial
- $h(n)$ le coût estimé pour arrivé au but à partir du nœud n

Il garde une liste des candidats, et en choisit succéssivement celui qui minimise le coût f et rajoute tous ses états adjacents (non visité) à la liste des candidats.

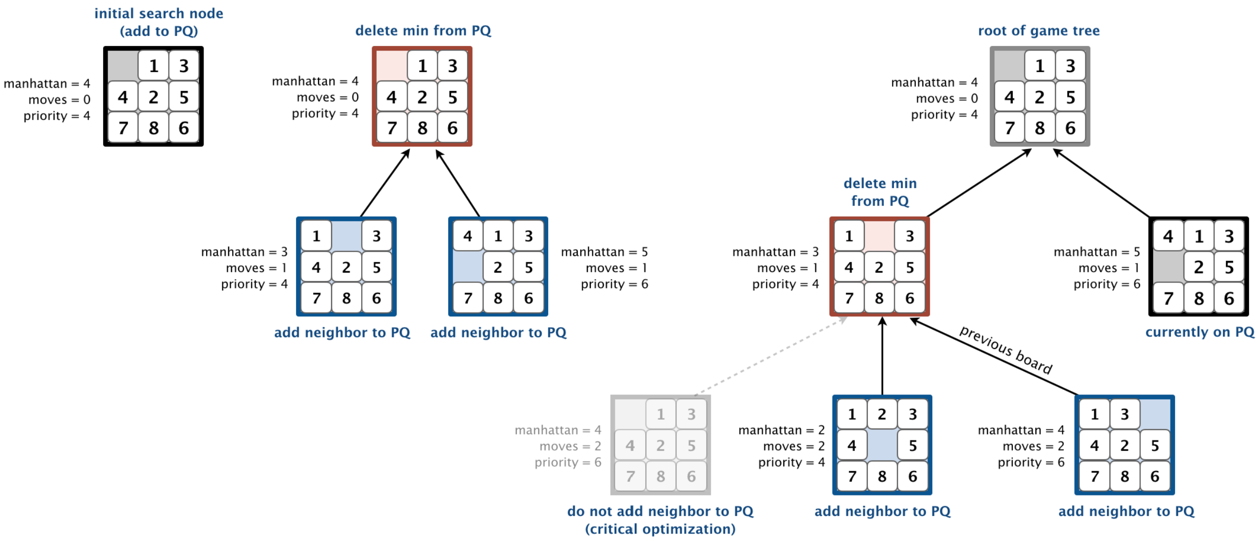


FIGURE 4 – Une partie de la graphe de résolution A*
(*source:* Princeton University - Computer Science Course COS226)

Les solutions obtenues sont optimale si l'heuristique est admissible, les meilleurs résultats sont obtenus pour la distance de Manhattan. (Wikipedia contributors 2019a)

Pour les configurations faciles la solution est obtenue instantanément, mais pour les configurations faciles l'algorithme peut prendre plus de temps que les algorithmes précédents. Cependant, pour les tailles plus grande que 3×3, A* est le meilleur algorithme pour trouver une solution.

3 Implémentation

Explication de notre code.

4 Conclusion

4.1 Développements possibles du projet

Ce qu'on peut faire encore.

4.2 Apport personnel

Ce qu'on a appris.

Références

Johnson, W. W., et W. E. Storey. 1879. « Notes on the 15-Puzzle ».

Reinefeld, Alexander. 1993. « Complete Solution of the Eight-Puzzle and the Benefit of Node Ordering in IDA * ».

Wikipédia. 2018. « Taquin — Wikipédia, l'encyclopédie libre ». http://fr.wikipedia.org/w/index.php?title=Taquin&oldid=153121375.

Wikipedia contributors. 2018. « Admissible heuristic — Wikipedia, The Free Encyclopedia ». https://en.wikipedia.org/w/index.php?title=Admissible_heuristic&oldid=873230067.

———. 2019a. « A* search algorithm — Wikipedia, The Free Encyclopedia ». https://en.wikipedia.org/w/index.php?title=A*_search_algorithm&oldid=925009518.

———. 2019b. « Iterative deepening depth-first search — Wikipedia, The Free Encyclopedia ». https://en.wikipedia.org/w/index.php?title=Iterative_deepening_depth-first_search&oldid=925129768.