



INSTITUT NATIONAL  
DES SCIENCES  
APPLIQUÉES  
**ROUEN NORMANDIE**

## CONTRAINTES ET PROGRAMMATION LOGIQUE

---

### Le Jeu du Taquin

---



Quentin LOISEAU  
Rand ASSWAD  
Génie Mathématique

*A l'attention de :*  
M. Habib ABDULRAB

## Table des matières

<b>1</b>	<b>Introduction</b>	<b>2</b>
1.1	Le jeu du taquin . . . . .	2
1.2	Complexité . . . . .	2
1.3	Le but du projet . . . . .	2
1.4	Formulation du problème . . . . .	2
<b>2</b>	<b>Algorithmes</b>	<b>3</b>
2.1	Depth-First Search (DFS) . . . . .	3
2.2	Heuristiques . . . . .	4
2.3	Algorithme Greedy . . . . .	5
2.4	Iterative Deepening DFS (ID-DFS) . . . . .	6
2.5	Algorithme A* . . . . .	6
<b>3</b>	<b>Implémentation</b>	<b>7</b>
3.1	Modèle du programme . . . . .	7
3.2	Algorithmes . . . . .	8
3.3	Utilisation du programme . . . . .	10
3.4	Quelques résultats . . . . .	10
<b>4</b>	<b>Conclusion</b>	<b>11</b>
4.1	Analyse des résultats . . . . .	11
4.2	Développements possibles du projet . . . . .	11
4.3	Apport personnel . . . . .	12
	<b>Références</b>	<b>12</b>

# 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  $3 \times 3$  ou  $4 \times 4$ , d'où l'appellation anglophone *8-puzzle* ou *15-puzzle* respectivement.

Le Rubik's Cube est considéré 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 et exige une résolution guidée afin d'atteindre avant d'épuiser les ressources (temps et mémoire).

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

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 1979) afin de montrer que la moitié des configurations initiales 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 porté 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 définit l'espace d'état  $E$  par toutes les configurations solvables du problème.

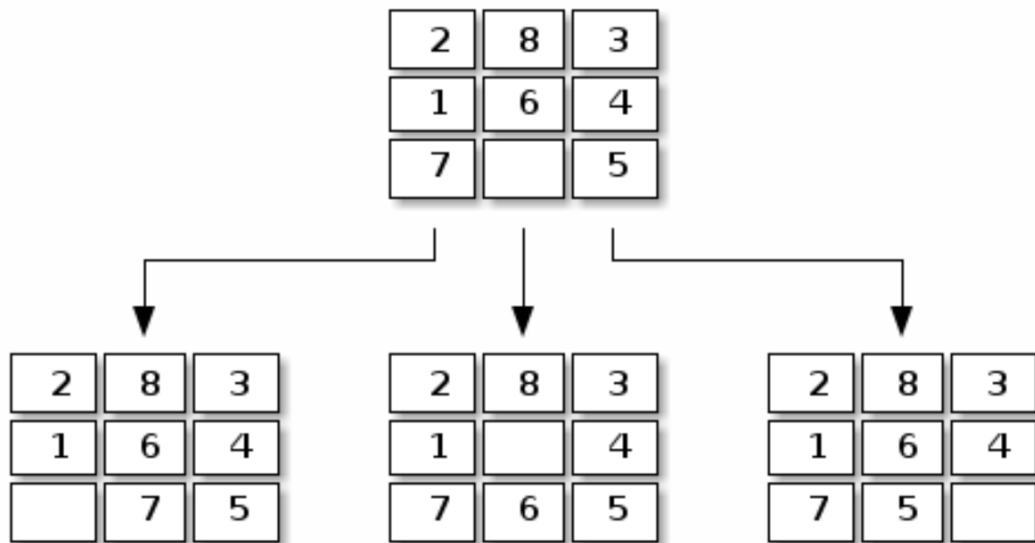


FIGURE 1 – Une partie du graphe du taquin

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).
```

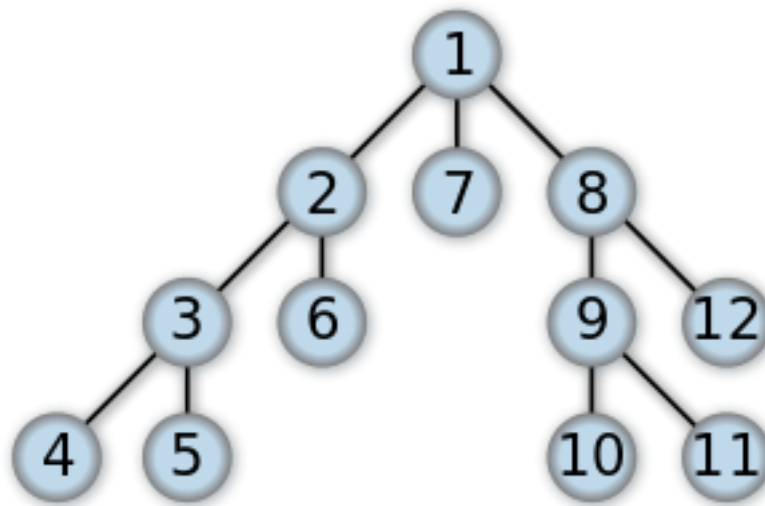


FIGURE 2 – L'ordre de parcours des nœuds dans l'algorithme DFS

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 précédant, 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  $h : E \rightarrow \mathbb{N}$  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)

$$d_{\text{Hamming}} : A^n \times A^n \rightarrow \mathbb{N}$$

$$(x, y) \mapsto \sum_{i=1}^n (1 - \delta_{x[i], y[i]}) = \begin{cases} 0 & \text{si } x[i] = y[i] \\ 1 & \text{sinon} \end{cases}$$

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  $n$ .

$$d_1 : V \times V \rightarrow \mathbb{R}_+$$

$$(x, y) \mapsto \sum_{i=1}^n |x_i - y_i|$$

Dans notre contexte, la distance est définie sur  $\{1, \dots, n\} \times \{1, \dots, m\}$  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			Heuristique	Hamming								Manhattan								$h^*$
8	1	3	1	2	3	Tuile	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8	
4		2	4	5	6	Distance	1	1	0	0	1	1	0	1	1	2	0	0	2	2	0	3	
7	6	5	7	8		Total	5								10								14

TABLE 1 – Heuristiques de Hamming et de Manhattan

Table 1 – Heuristiques de Hamming et de Manhattan source: Princeton University - Computer Science Course COS226

## 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  $3 \times 3$ , 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 successivement 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.

### Complexité

- Complexité en temps:  $O(b^d)$
- Complexité spacia:  $O(d)$

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  $3 \times 3$ , 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 complexité spacia 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 successivement celui qui minimise le coût  $f$  et rajoute tous ses états adjacents (non visité) à la liste des candidats.

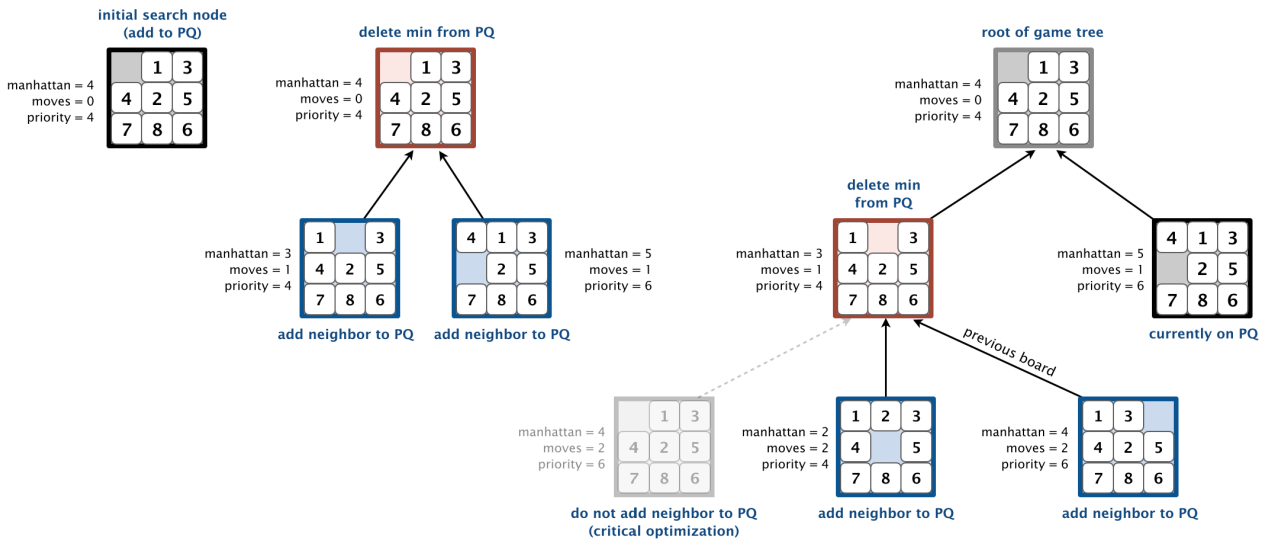


FIGURE 3 – Une partie de la graphe de résolution A\*  
source: Princeton University - Computer Science Course COS226

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 difficiles l’algorithme peut prendre plus de temps que les algorithmes précédents. Cependant, pour les tailles plus grande que  $3 \times 3$ , A\* est le meilleur algorithme pour trouver une solution.

## 3 Implémentation

### 3.1 Modèle du programme

Nous avons implémenté notre programme sous l’environnement SWI Prolog qui est complet, libre, bien documenté, et permet de construire des programmes modulaires.

Notre programme donc est modulaire et réutilisable.

```
taquin
├── util
│   ├── core.pl.....core predicates
│   ├── hamming.pl.....hamming heuristic definition
│   ├── heuristic.pl.....heuristics wrapper predicates
│   ├── manhattan.pl.....manhattan heuristic definition
│   ├── moves.pl.....states adjacency
│   └── test.pl.....testing tools
├── taquin_astar.pl.....A* algorithm
├── taquin_dfs.pl.....DFS-style algorithms
├── taquin.pl.....taquin solver wrapper
├── test_3x3.pl.....3×3 example puzzles
├── test_3x4.pl.....3×4 example puzzles
├── test_4x4.pl.....4×4 example puzzles
└── README.md.....quick usage guide
```

Nous avons implémenté le jeu taquin  $n \times m$  à l’aide du prédicat `dim/2` qui prend en argument le nombre de lignes et le nombre des colonnes. La base de connaissance doit contenir une et seulement une dimension.

Nous fournissons dans le module `util/core` le prédicat `goal/1` qui définit l’état objectif pour la dimension prédéfinie dans la base de connaissance. Le module `core` dépend des modules `util/moves` et `util/heuristic`.

Le module `util/moves` définit l’adjacente à l’aide du prédicat `move/3`

```
move(Before, After, Direction)
```

qui est vrai si en glissant une tuile de l’état `Before` dans la direction `Direction` l’état `After` est obtenu.

Le module `util/heuristic` fournit le prédicat `h/2` qui calcule la fonction heuristique d’un état donné, la fonction heuristique à utiliser doit être définie dans la base de connaissance à l’aide du prédicat `heuristic/1` qui accepte les valeurs `manhattan`, `hamming` et `m3h`. Ce module dépend des modules `util/manhattan` et `util/hamming`.

Le module `taquin_dfs` fournit les prédicats des algorithmes basés sur le principe DFS, et le module `taquin_astar` fournit le prédicat de l’algorithme A\*. Ces modules dépendent du module `util/core` (qui inclut les modules nécessaires).

Le module `taquin` fournit un *wrapper* pour tous les algorithmes implémentés dans les prédicats `solve/3` et `solve/4`.

Utilisation:

```
solve(+Puzzle, -Solution, +Algorithm, +Heuristic).
```

- `+Puzzle`: une configuration initiale donnée
- `-Solution`: un chemin (liste des états) du but jusqu’à `Puzzle`
- `+Algorithm`: l’algorithme à utiliser (`dfs`, `iddfs`, `greedy` ou `astar`)
- `+Heuristic`: heuristique à utiliser (`manhattan`, `hamming` ou `m3h`)

Le prédicat `solve/3` choisit lui-même l’heuristique en fonction de l’algorithme choisi (recommandé).



## 3.2 Algorithmes

### 3.2.1 Depth-First Search

La définition du prédicat `dfs` est similaire à celle présenté dans l'explication de l'algorithme en y rajoutant la contrainte `\+member(E, Chemin)` afin d'éviter les problèmes dont nous avons parlé.

```
% dfs(Etat, CheminEnCours, CheminFinal)
dfs(Initial, Path, Path) :- goal(Initial).      % condition d'arrêt
dfs(Initial, Visited, Path) :-
    move(Initial, State, _),                    % un mouvement possible
    \+member(State, Visited),                  % l'état n'existe pas dans le chemin
    dfs(State, [State|Visited], Path).          % reste du chemin

% dfs/2: wrapper pour dfs/3 avec chemin vide
dfs(Initial, Path) :- dfs(Initial, [], Path).
```

### 3.2.2 Iterative Deepening DFS

La définition d'une itération de ID-DFS est similaire à celle de DFS, en y rajoutant la contrainte de profondeur.

```
% iddfs(Etat, CheminEnCours, CheminFinal, Profondeur)
iddfs(Initial, Path, Path, _) :- goal(Initial). % condition d'arrêt
iddfs(Initial, Visited, Path, Depth) :-
    Depth > 0,                                % profondeur non nul
    move(Initial, State, _),                  % un mouvement possible
    \+member(State, Visited),                % l'état n'existe pas dans le chemin
    D1 is Depth - 1,                         % decrementer le profondeur
    iddfs(State, [State|Visited], Path, D1).  % reste du chemin
```

Il reste d'initialiser la boucle avec un profondeur maximal sous-estimé par une heuristique admissible

```
% iddfs/2 initialise la boucle avec prondeur max = h(état initial)
iddfs(Initial, Path) :- h(Initial, H), iddfs(Initial, Path, H), !.

% iddfs/3 wrapper d'une itération de iddfs de profondeur donné
iddfs(Initial, Path, Depth) :- iddfs(Initial, [], Path, Depth).
iddfs(Initial, Path, H) :- H1 is H + 1, iddfs(Initial, Path, H1).
```

### 3.2.3 Algorithme Greedy

La définition de l'algorithme Greedy est similaire à celui de DFS, en prenant le meilleur mouvement (localement) non visité.

```
% greedy(Etat, CheminEnCours, CheminFinal)
greedy(Initial, Path, Path) :- goal(Initial).  % condition d'arrêt
greedy(Initial, Visited, Path) :-
    bestMove(Initial, Visited, State),          % le meilleur mouvement
    greedy(State, [State|Visited], Path).        % reste du chemin

% greedy/2: wrapper pour greedy/3 avec chemin vide
greedy(Initial, Path) :- greedy(Initial, [], Path).
```

Le prédicat `greedy/3` se définit grace aux prédicats auxiliaires

```

% cheapest/2 trouve l'état avec l'heuristique la plus petite
cheapest([State], State).
cheapest([A, B|T], M) :- h(A, Ac), h(B, Bc), Ac <= Bc, cheapest([A|T], M).
cheapest([A, B|T], M) :- h(A, Ac), h(B, Bc), Ac > Bc, cheapest([B|T], M).

% bestMove/3 trouve le voisin non-visité le moins cher
bestMove(State, Visited, Next) :-
    % construire la liste des voisins non-visités
    findall(
        Neighbor,
        (move(State, Neighbor, _), \+member(Neighbor, Visited)),
        Neighbors
    ),
    % en choisir le voisin qui minimise l'heuristique
    cheapest(Neighbors, Next).

```

### 3.2.4 Algorithme A\*

L'algorithme A\* est le plus complexe parmi les algorithmes présentés, contrairement aux autres algorithmes, il garde une liste des candidats qui ne cesse d'augmenter jusqu'à ce qu'un chemin est trouvé.

Les algorithmes de type DFS gardent également une liste de candidats implicite: **une pile**, c'est-à-dire une liste LIFO (last in first out). En revanche, avec l'algorithme A\* il faut gérer la liste des candidats explicitement, afin d'optimiser le temps de calcul nous utilisons un **Priority Queue** (file de priorité), qui est une liste d'éléments ordonnés dans l'ordre (croissant ou décroissant) par rapport à une fonction de priorité.

Notre fonction de priorité est  $f(n) = g(n) + h(n)$  que nous souhaitons minimiser, notre Priority Queue est donc dans l'ordre croissant par rapport à  $f$ .

Tout d'abord, nous définissons le terme **node/3** qui représente un nœud du graphe de résolution de A\*. Le terme **node(S, P, F)** se définit par:

- **Etat S**: une configuration du puzzle.
- **Chemin P**: le chemin pour arriver à S.
- **Coût F**: la fonction  $f$  évaluée en S par rapport au chemin P.

Au lieu d'avoir à trier la liste, nous avons implémenté les prédicats **pushQueue/3** et **appendQueue/3** qui permettent d'insérer des éléments dans le Queue dans le bon placement directement.

```

% pushQueue(+Node, +Queue, -NewQueue)
% permet de rajouter un nœud au Queue
pushQueue(Node, [], [Node]) :- !. % queue null
pushQueue(node(S1,P1,F1), [node(S2,P2,F2)|Q], [node(S1,P1,F1),node(S2,P2,F2)|Q]) :-
    F1 <= F2, % le nouveau nœud est le moins cher
    !. % arrêter la recherche
pushQueue(node(S1,P1,F1), [node(S2,P2,F2)|Q0], [node(S2,P2,F2)|Q]) :-
    pushQueue(node(S1,P1,F1), Q0, Q).

% appendQueue(NodeList, Queue, NewQueue)
% permet de rajouter une liste de nœuds non-ordonnés au Queue
appendQueue([], Q, Q). % liste nulle
appendQueue([Node|T], Q0, Q) :-
    pushQueue(Node, Q0, Q1), % rajouter le premier
    appendQueue(T, Q1, Q). % rajouter le reste

```

Chaque itération de A\* consiste à générer les enfants du nœud prioritaire et de les rajouter au Queue.

```

% astar_search(NodeQueue, Path, Solution)
astar_search([node(Goal, Path, _)|_], _, Path) :- goal(Goal).
astar_search([node(S, P, H)|Queue], Visited, Solution) :-
    % générer les voisins de S

```

```

findall(Neighbor, move(S, Neighbor, _), Sneighbors),
% générer une liste des enfants
nodeChildren(node(S, P, H), Sneighbors, Visited, Children),
% rajouter les enfants au Queue
appendQueue(Children, Queue, SortedQueue),
% recommencer l'itération avec le nouveau Queue
astar_search(SortedQueue, [S|Visited], Solution).

```

Le prédicat `nodeChildren/4` permet de générer les enfants d'un nœud à partir de la liste de ses voisins.

```

% nodeChildren(ParentNode, CandidateChildren, Visited, Children)
nodeChildren(_, [], _, []) :- !. % pas de candidats
nodeChildren(node(Parent, Path, ParentF), [Child|Others], Visited,
  [node(Child, [Child|Path], F)|Children]) :-
  % l'état n'existe pas dans le chemin
  \+member(Child, Visited),
  % calculer le coût de l'enfant
  length(Path, G1), h(Child, H), F is G1 + 1 + H,
  % générer les enfants pour le reste des voisins
  nodeChildren(node(Parent, Path, ParentF), Others, [Child|Visited], Children),
  !. % ignorer la règle suivante et passer aux autres candidats
nodeChildren(Node, [_|Others], Visited, Children) :- % l'état existe dans le chemin
  nodeChildren(Node, Others, Visited, Children). % passer aux autres candidats

```

Il ne reste que de lancer l'algorithme avec le queue contenant l'état initial dont le coût est  $f(n_0) = h(n_0)$ .

```

astar(Initial, Path) :-
  h(Initial, H), % évaluer f(n) = h(n)
  astar_search([node(Initial, [], H)], [Initial], Path). % lancer A*

```

### 3.3 Utilisation du programme

Nous avons fourni le module `util/test` qui permet de tester le programme, le module définit les prédicats

- `printState/1`: affiche un état comme une grille
- `printSolution/2`: affiche un chemin
- `testPuzzle/3`, `testPuzzle/2`, `testPuzzle/1`: wrapper pour résoudre et afficher la solution (les 2<sup>ème</sup> et 3<sup>ème</sup> arguments sont optionnels)
- `debugPuzzle/3`: affiche le temps de résolution et longueur du chemin

Nous fournissons quelques exemples de puzzles dans les fichiers `test_3x3.pl`, `test_3x4.pl` et `test_4x4.pl`, à l'aide de `term puzzle(Puzzle, Difficulty)`.

Pour utiliser notre programme, lancer l'un de nos programmes tests

```
$ swipl test_3x3.pl
```

et lancer la requête suivante:

```

% Algorithm = dfs, iddfs, greedy ou astar
% Heuristic = manhattan, hamming ou m3h
?- puzzle(Puzzle, Difficulty),
  Algorithm = astar, Heuristic = manhattan,
  testPuzzle(Puzzle, Algorithm, Heuristic).

```

### 3.4 Quelques résultats

Nous avons testé nos algorithmes sur quelques exemples représentatifs, voici les résultats trouvés pour les puzzles de taille  $3 \times 3$ .

État	Difficulté	Algorithmes			Heuristique		
		Greedy	ID-DFS	A*	Manh.	Ham.	M3H
1 2 3 4 5 7 8 6	triviale	$t = 13\text{ms}$ $N = 1$	$t = 3\text{ms}$ $N = 1$	$t = 9\text{ms}$ $N = 1$	1	1	4
1 3 4 2 5 7 8 6	facile	$t = 18\text{ms}$ $N = 4$	$t = 6\text{ms}$ $N = 4$	$t = 19\text{ms}$ $N = 4$	4	4	16
8 1 3 4 2 7 6 5	moyenne	$t = 96\text{ms}$ $N = 34$	$t = 237\text{ms}$ $N = 14$	$t = 210\text{ms}$ $N = 14$	10	5	25
4 3 8 2 1 6 5 7	difficile	$t = 418\text{ms}$ $N = 176$	$t = 6\,916\text{ms}$ $N = 20$	$t = 587\text{ms}$ $N = 20$	16	8	40
6 4 7 8 5 3 2 1	maximale	$t = 108\text{ms}$ $N = 45$	$t = 9\,233\text{ms}$ $N^* = 41$	$t = 27\,005\text{ms}$ $N = 31$	21	7	42
8 6 7 2 5 4 3 1	maximale	$t = 644\text{ms}$ $N = 243$	$t = 23\,124\text{ms}$ $N^* = 41$	$t = 21\,933\text{ms}$ $N = 31$	21	7	42

TABLE 2 – Comparaison des algorithmes sur des taquin de taille  $3 \times 3$ Table 2 – Comparaison des algorithmes sur des taquin de taille  $3 \times 3$ 

Les états classés de difficulté *maximale* sont les configurations avec les chemins les plus longue pour un puzzle  $3 \times 3$ . (Reinefeld 1993)

Les résultats du DFS sont absents du tableau car pour tous états non triviaux l'algorithme ne trouve pas un chemin avant qu'on perde notre patience.

Les solutions de Greedy sont calculé avec l'heuristique local **m3h**. Les solutions de ID-DFS et A\* sont trouvés avec l'heuristique de Manhattan à part ceux marquées avec un asterisk; elles sont trouvées avec **m3h** qui n'est pas admissible, car l'algorithme met beaucoup de temps pour trouver le chemin avec Manhattan et nous n'avons jamais réussi à obtenir un résultat. Néanmoins, si on part directement de l'itération de profondeur 31, les chemins optimaux sont trouvés en  $237\,514\text{ms} \approx 4\text{min}$  et  $271\,363\text{ms} \approx 4.5\text{min}$  respectivement.

## 4 Conclusion

### 4.1 Analyse des résultats

Nous avons trouvé conformément à nos attentes que l'algorithme A\* est le mieux adapté à ce problème. Les autres algorithmes peuvent être plus rapides et/ou chanceux dans les cas simples, mais dans les cas les plus complexes et surtout en dimensions plus grandes, A\* est le plus efficace et le meilleur candidat pour trouver une solution en temps raisonnable.

Néanmoins, l'optimisation locale de Greedy le rend un candidat respectable pour trouver un chemin rapidement, ce que nous avons présenté pendant notre soutenance. En effet, pour l'une des configurations  $3 \times 4$  que nous avons testé Greedy était le seul à nous donner une réponse (en temps raisonnable).

### 4.2 Développements possibles du projet

#### 4.2.1 Algorithme IDA\*

Malgré la lenteur de l'algorithme ID-DFS, son idée reste très intéressante. Sa variante **Iterative Deepening A\* (IDA\*)** (Korf 1985) emprunte la fonction coût de A\*  $f(n) = g(n) + h(n)$  et donne des solutions plus rapidement que A\* et en économisant beaucoup de mémoire par rapport à ce dernier. Il sera certainement plus intéressant d'étudier et d'implémenter l'algorithme IDA\*.

### 4.2.2 Mouvements

Nous avons implémentés une grille de taille dynamique, ce qui rend le projet plus intéressant. Néanmoins, chaque appelle au prédicat `move/3` nécessite un calcul non-négligeable dans les cas complexes, il est ainsi important de définir les mouvements *à la main* pour les tailles  $3 \times 3$  et  $4 \times 4$ .

### 4.2.3 Portabilité

Les algorithmes utilisés dans ce projet peuvent s'appliquer à la résolutions de différents problèmes (e.g. Rubik's Cube, Flowshop, etc). Il sera certainement intéressant d'adapter l'implémentation afin d'avoir un code compatibles avec plusieurs instances de problèmes de graphe.

## 4.3 Apport personnel

Ce projet a été une excellente opportunité pour approfondir notre connaissance en IA et notre maîtrise de la programmation logique.

## Références

Johnson, W. W., et W. E. Storey. 1979. « Notes on the 15-Puzzle ».

Korf, Richard. 1985. « Depth-First Iterative-Deepening: An Optimal Admissible Tree Search ». [https://cse.sc.edu/~mgv/csce580f09/gradPres/korf\\_IDAStar\\_1985.pdf](https://cse.sc.edu/~mgv/csce580f09/gradPres/korf_IDAStar_1985.pdf).

Reinefeld, Alexander. 1993. « Complete Solution of the Eight-Puzzle and the Benefit of Node Ordering in IDA \* ». <https://www.ijcai.org/Proceedings/93-1/Papers/035.pdf>.

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](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](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](https://en.wikipedia.org/w/index.php?title=Iterative_deepening_depth-first_search&oldid=925129768).