

Université des Sciences et de la Technologie Houari Boumedien
Faculté d'Electronique et d'Informatique
Département d'informatique



USTHB

Master I Système Informatique Intelligents

Module : Méta-heuristique & algorithmes évolutionnaires

Projet TP

Résolution du problème de taquin (Approche
par espace des états)

Réalisé par :

BELLAZRAG Nadia 181831043933

ELFANI Rima 171732058007

GUETTAF Sarah 161631039302

LAMINI Narimene 181831043933

Année universitaire : 2021-2022

Sommaire

Liste des tableaux

Table de figures

| | |
|---|-----------|
| Représentation du problème et définitions | 6 |
| 1.1 Problématique..... | 6 |
| 1.2 Définitions..... | 6 |
| 1.2.1 Problème du taquin..... | 6 |
| 1.2.2 Stratégie de recherche dans l'espace des états..... | 7 |
| Implémentation des méthodes constructives pour le taquin | 11 |
| 2.1 Structure de données..... | 11 |
| 2.1.1 État..... | 11 |
| 2.1.2 Solution..... | 11 |
| 2.2 Conception et pseudo code..... | 13 |
| 2.2.1 Gestion de la liste Ouverte | 14 |
| 2.2.2 Complexité théorique temporelle est spatiale | 16 |
| Expérimentations..... | 17 |
| 3.1 Environnement de travail..... | 17 |
| 3.2 Outils utilisés | 17 |
| 3.2.1 Langage de programmation..... | 17 |
| 3.2.2 IDE | 17 |
| 3.3 Interface graphique..... | 18 |
| 3.3.1 Détails..... | 19 |
| 3.3.2 Jeu de test..... | 19 |
| 3.4 Analyse, interprétation et comparaison des résultats..... | 23 |
| 3.2.1 Instances..... | 23 |
| 3.2.2 Résultats..... | 24 |
| 3.5 Statistiques..... | 32 |

| | |
|----------------------------|----|
| 3.6 Étude comparative..... | 34 |
| Conclusion..... | 35 |
| Bibliographie..... | 36 |

Liste des tableaux

| | |
|---|----|
| Table 2.1- Définition des heuristiques..... | 14 |
| Table 2.2- Complexité théorique temporelle et spatiale | 15 |
| Table 3.1- Détails sur les différentes sections de l'interface | 18 |
| Table 3.2- Résultats d'exécution en largeur d'abord..... | 24 |
| Table 3.3- Résultats d'exécution en profondeur d'abord..... | 25 |
| Table 3.4- Résultats d'exécution en profondeur d'abord (seuil = 50)..... | 26 |
| Table 3.5- Résultats d'exécution en profondeur d'abord (seuil = 80)..... | 27 |
| Table 3.6- Résultats d'exécution en profondeur d'abord (seuil = 100)..... | 27 |
| Table 3.7- Résultats d'exécution de l'algorithme A*(Distance Euclidienne)..... | 28 |
| Table 3.8- Résultats d'exécution de l'algorithme A*(Distance de Manhattan)..... | 30 |
| Table 3.9- statistiques | 30 |

Table des figures

| | |
|---|----|
| Figure 1.1 - Exemple du taquin 3x3 | 7 |
| Figure 1.2 - Parcours des nœuds selon l'ordre DFS et BFS..... | 8 |
| Figure 3.1 - Performances de la machine..... | 16 |
| Figure 3.2 - Interface graphique du jeu de taquin..... | 17 |
| Figure 3.3 - Etat initial d'une instance de taquin..... | 19 |
| Figure 3.4 - Exécution (DFS)..... | 19 |
| Figure 3.5 - Exécution (BFS)..... | 20 |
| Figure 3.6 - Exécution (A*- distance Euclidienne) | 20 |
| Figure 3.7 - Exécution (A*- distance de Manhattan)..... | 21 |
| Figure 3.8 - Instance insoluble..... | 21 |

Chapitre 1

Représentation du problème et définitions

1.1 Problématique

Dans ce projet, nous allons aborder un problème d'un jeu tant de fois étudié avec différentes approches, n'est autre que le « Jeu du taquin ». Ce jeu est devenu très familier et populaire à la fin du siècle dernier.

Notre étude dans la première partie de ce projet va se focaliser sur l'implémentation des solutions pour le jeu du taquin 3x3 à l'aide des méthodes aveugles dites aussi à base d'espace d'états ou méthodes exactes, et des méthodes heuristiques dites méthodes guidées. On s'intéressera à la production d'une solution dite minimale.

1.2 Définitions

Dans ce qui suit, nous allons représenter notre problème du **jeu du taquin** et donner quelques définitions sur les méthodes de sa résolution.

1.2.1 Problème du taquin

Le jeu de taquin consiste à manipuler une table carrée T à $n \times n$ cases, avec $n \geq 2$, sur laquelle on pose $n^2 - 1$ pièces numérotées de 1 à $n^2 - 1$. Il reste donc une seule case vide appelée trou. Le seul mouvement ou coup autorisé consiste à faire glisser l'une des pièces adjacentes au trou vers celui-ci, ce qui revient à échanger leurs positions respectives [1]. En d'autres termes, le jeu vise à déplacer les jetons un par un pour transformer un "état initial" quelconque en un "état but" fixé, **en un nombre minimal de coups** [2]. La figure 1.1 montre un exemple de taquin appelé « Taquin 3x3 » ayant un état initial (E.I) et un état but (E.B).



1.2.2 Stratégie de recherche dans l'espace des états

Afin de résoudre des problèmes de l'IA de manière classique, on devra tout au début simuler le problème de façon symbolique, puis on entamera la création de l'espace et enfin on utilisera des méthodes et des stratégies pour trouver des solutions dans notre espace créé.

Les éléments qui suivent représentent les étapes et les notions à définir pour la résolution du problème : [3]

- **Etat du problème** : L'état de configuration que prend ce dernier lors de la transition d'un état 'i' vers l'état 'j'. Dans notre étude, c'est l'état du taquin.
- **Etat initial** : C'est le premier état à partir duquel l'arbre de recherche commence à se développer.
- **Etat final** : dit aussi l'état but ou l'état cible. Une fois trouvé, le problème est résolu.
- **Les opérateurs entre états** : Peuvent être définies comme étant un outil de transition d'un état à un autre, dans notre cas du jeu de taquin n*n, on devra manipuler le trou en le déplaçant comme suit :

- ↙ Déplacer le trou à gauche
 ↘ Déplacer le trou à droite
 ↑ Déplacer le trou vers le haut
 ↓ Déplacer le trou vers le bas

La figure suivante montre une partie de résolution d'une instance du taquin en utilisant les opérateurs et les notions définies précédemment :

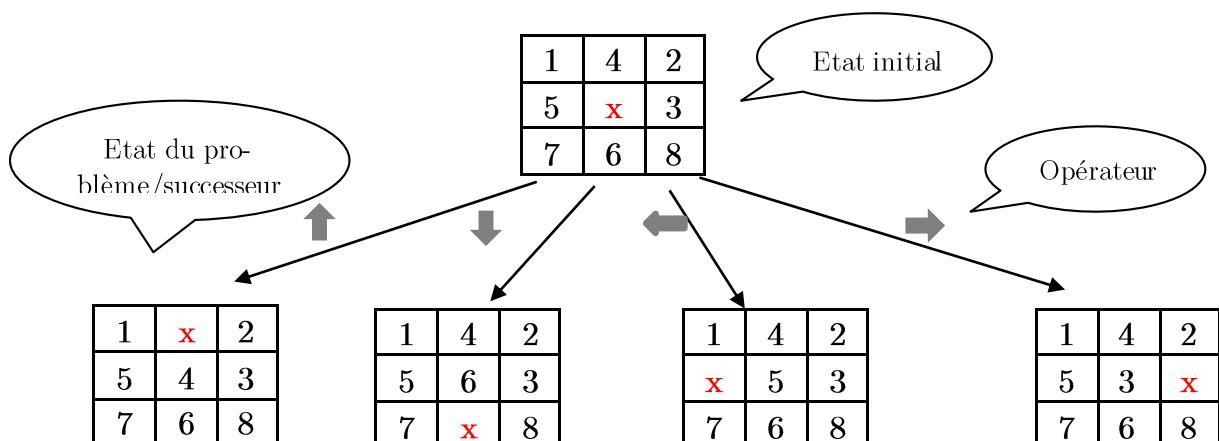


Figure 1.1 - une partie de résolution d'une instance du taquin 3x3

Comme on peut le constater, cette succession d'opérations engendre la construction d'un « Arbre de recherche » ayant comme nœud “Etat initial” et comme successeur “Etat du problème” généré. Ces opérations se répéteront jusqu'à ce qu'on arrive à l'état but, et la solution sera le chemin parcouru de l'état initial vers l'état final trouvé par la stratégie de recherche.

Nous pouvons classer les différentes stratégies de recherche en deux grandes catégories :

1. Stratégies de recherche aveugles

Ces méthodes utilisent des algorithmes non informés, c'est-à-dire qui réalisent une recherche exhaustive en passant par toutes les solutions et en les testant une à une. Pour notre projet, nous allons aborder deux méthodes de recherche aveugles, à savoir :

➤ Recherche en profondeur d'abord (Depth-First-Search/ DFS)

La recherche en profondeur signifie le développement d'une branche entière avant de parcourir le reste de l'arbre en effectuant du "backtracking", ce développement peut amener à trois situations différentes [4] :

- La solution est trouvée et dans ce cas le développement de la branche s'arrête avec une possibilité de "backtraking" si d'autres solutions sont encore sollicitées.
- La solution n'est pas trouvée et un état d'échec est détecté (c'est un état qui n'engendre pas d'autres états) et dans ce cas le "backtraking" est appliqué pour poursuivre la recherche.
- Une branche infinie est à explorer et dans ce cas, un test d'arrêt sur une profondeur maximale doit être appliquée.

Les performances de la recherche en profondeur sont liées à l'ordre d'exploration des branches de l'arbre. Autrement dit, si la solution se trouve dans la première branche à explorer cette stratégie devient optimale.

➤ Recherche en largeur d'abord (Breadth-First-Search/ BFS)

La recherche en largeur signifie que les états doivent être visités en parcourant l'arbre niveau par niveau, autrement dit, tous les successeurs d'un état donné sont visités l'un après l'autre avant le passage au niveau suivant. Pour effectuer le parcours en largeur, une file est utilisée. Le parcours s'arrête quand un état final est trouvé ou quand une profondeur maximale est atteinte. Ce parcours est très couteux en espace mais il garantit de trouver la

solution si elle existe, tandis que le parcours en profondeur peut ne pas converger même si la solution existe à cause d'une branche infinie. [4] La figure 1.2 illustre les deux méthodes DFS et BFS :

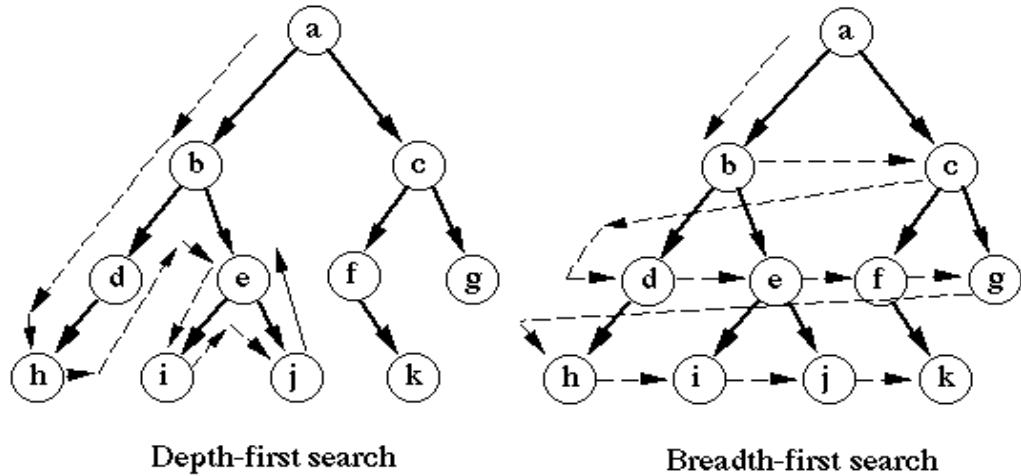


Figure 1.2 - Parcours des nœuds selon l'ordre DFS et BFS

Si on applique ces méthodes aveugles sur une grande catégorie de problèmes, on constatera qu'elles sont peu efficaces par rapport au temps d'exécution limité ainsi qu'à l'espace mémoire. Donc, pour accélérer la recherche on adoptera des méthodes guidées, connues par « Approches heuristiques ».

2. Stratégies de recherche guidées (Heuristiques)

Une heuristique est une technique qui améliore l'efficacité d'un processus de recherche, en sacrifiant éventuellement l'exactitude ou l'optimalité de la solution. Pour des problèmes d'optimisation (NP-complets) où la recherche d'une solution exacte (optimale) est difficile (coût exponentiel), on peut se contenter d'une solution satisfaisante donnée par une heuristique avec un coût plus faible. Certaines heuristiques sont polyvalentes (elles donnent d'assez bons résultats pour une large gamme de problèmes) alors que d'autres sont spécifiques à chaque type de problème.[5]

Les heuristiques permettent d'accélérer la recherche dans des domaines applicatifs spécifiques. Différentes heuristiques de parcours sont suffisamment générales pour être appliquées à plusieurs catégories de problèmes d'optimisation combinatoire, elles portent le nom de mét-heuristique. [6] Dans ce projet, nous allons implémenter une des méthodes guidées pour résoudre le problème de taquin, c'est l'algorithme A*.

➤ Algorithme A*

L'algorithme A* effectue un parcours en largeur guidé par une heuristique définie comme suit [4] :

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

Où :

$g(x)$ représente le coût du chemin entre la racine et x

$h(x)$ représente une estimation du coût restant entre x et un éventuel sommet solution (BUT) accessible à partir de x .

Ainsi $f(x)$ serait une estimation du coût d'un chemin entre la racine et le BUT passant par x . Pour que le chemin trouvé soit toujours optimal, on impose à $h(x)$ de ne jamais surestimer le coût du trajet restant réel.

Toute heuristique qui sous-estime la réalité est une heuristique qui permet de trouver le chemin optimal. Nous disons que h_2 est plus informée que h_1 si toutes les deux sont minorants et si $h_2(e) > h_1(e), \forall e \in S$. En effet h_2 permet de trouver le chemin optimal plus rapidement car elle est plus proche de $h(e)$.

Chapitre 2

Implémentation des méthodes constructives pour le taquin

2.1 Structure de données

Dans cette section, nous allons présenter les structures de données utilisées pour le stockage et la manipulation des états du taquin ainsi que la représentation de la solution (l'ensemble des états du chemin reliant l'état initial et l'état but).

2.1.1 État

Comme nous avons vu dans le chapitre précédent, un état du taquin est une grille $n \times n$ dont chaque case représente un entier de 1 à 8 et une seule case vide. Pour représenter l'ensemble des états, nous avons besoin d'une structure de données qui nous permet de manipuler l'information facilement et légèrement. Nous allons donc représenter notre taquin par une matrice de dimension identique à celle d'un état du taquin (dans notre projet, nous allons étudier le taquin 3×3). Les éléments de cette matrice sont de même, des entiers de 1 à 8, on représente la case vide par le chiffre 0.

L'exemple suivant montre une instance du problème du taquin et sa représentation matricielle :

| | | |
|---|---|---|
| 1 | 4 | 2 |
| 5 | x | 3 |
| 7 | 6 | 8 |

Représentation matricielle

$$M = \begin{bmatrix} 1 & 4 & 2 \\ 5 & 0 & 3 \\ 7 & 6 & 8 \end{bmatrix}$$

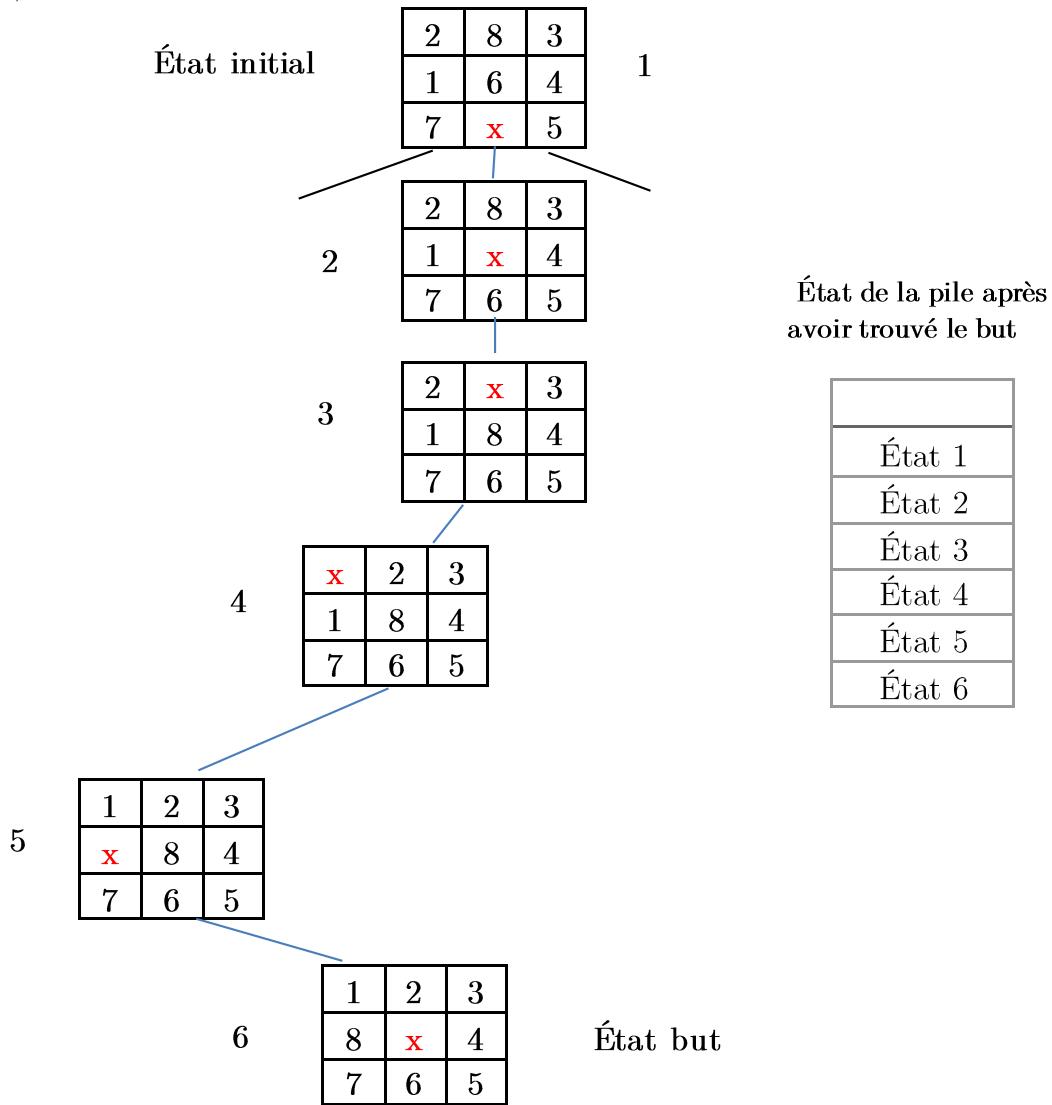
2.1.2 Solution

La solution d'une instance du taquin est un ensemble des états reliant l'état initial à l'état cible. En d'autres termes, c'est le chemin que l'algorithme a traversé pour arriver au but. Nous avons besoin d'une structure de donnée qui nous permet de stocker l'ensemble des états constituant ce chemin. Pour ce faire, nous allons utiliser une pile. Le premier élément empilé dans cette pile est l'état but

que l'algorithme a trouvé, le prochain élément à empiler est le parent de l'état but. Sachant que chaque état est relié avec son parent, ce processus d'empilement se répètera jusqu'à arriver à l'état initial (car ce dernier n'a pas de parent).

Ensuite, pour bien visualiser le chemin qui nous a mené vers le but, nous allons dépiler les états qui se trouvent dans pile où l'état initial se trouve au sommet de la pile et l'état but sera le dernier élément à dépiler.

L'exemple suivant montre une instance du taquin et sa solution (le chemin de 1 à 6)



Les états sont empilés dans l'ordre inverse, en défilant ces éléments, nous pourrons bien récupérer la solution de notre problème.

2.2 Conception et pseudo code

L'algorithme suivant est commun entre les différentes méthodes de recherche avec graphe. La seule différence réside dans la manière de réordonner les nœuds dans la liste ouverte. (Ligne 8 de l'algorithme) [5]

Algorithme : Algorithme de recherche avec graphe

Variables

2 Listes **Fermée** et **Ouverte** initialement vides : Fermée contient les nœuds qui ont été déjà développés et Ouverte contient les nœuds feuilles qui ne sont pas encore développés

DEBUT

1) Créer un graphe de recherche G qui consiste uniquement en nœud de départ d,

Mettre d sur une liste appelée **Ouverte**.

2) Créer une liste appelée **Fermée** qui est initialement vide.

3) BOUCLE : SI **Ouverte** est vide alors

 ECHEC

 Exit.

4) Sélectionner le 1er nœud de **Ouverte**, l'enlever de **Ouverte** et le mettre dans FERME, appeler ce nœud n.

5) Si n est un nœud But alors

 Terminer avec succès. Renvoyer le chemin obtenu le long des pointeurs (construits en phase 7) de n jusqu'à d dans G.

 Exit.

6) Si n n'a pas été déjà développé alors

 Développer le nœud n, produisant l'ensemble M de ses successeurs et les mémoriser comme successeurs de n.

7) Mémoriser un pointeur vers n à partir des éléments de M.

 Ajouter ces éléments de M à **Ouverte**

8) Réordonner **Ouverte**, soit arbitrairement soit selon une heuristique Fsi.

9) Aller à Boucle.

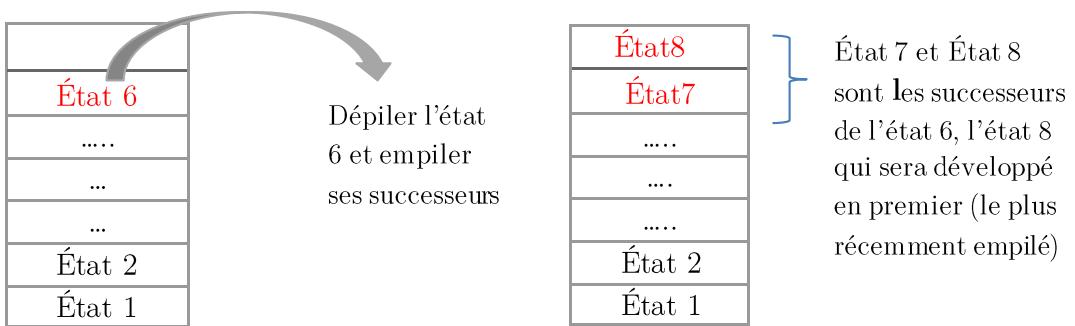
FIN.

Les nœuds sont ordonnés en phase 8 pour sélectionner les meilleurs pour être développés en phase 4 : Cet ordre peut être **arbitraire** (aveugle) ou suivant **une heuristique**.

2.2.1 Gestion de la liste Ouverte

➤ Recherche en profondeur d'abord (DFS)

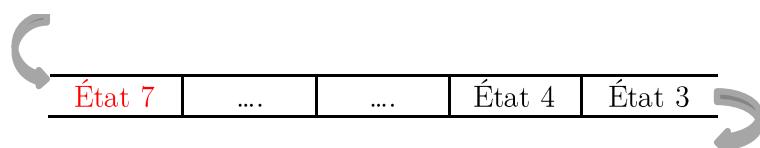
Cette méthode de recherche consiste à manipuler les nœuds de l'arbre en utilisant une pile (la liste ouverte est une pile). Le nœud à développer sera donc le plus récemment empilé et qui a la profondeur la plus élevée (dernier arrivé, premier servi), ce qui rend l'arbre développé en profondeur d'abord.



➤ Recherche en profondeur d'abord (BFS)

Contrairement à la recherche en profondeur d'abord, les nœuds sont visités de tel sorte à parcourir l'arbre niveau par niveau, cela peut être réalisé par la sélection du nœud le moins récemment insérer dans ouverte, d'où une gestion LIFO de la liste.

L'état 7 est récemment enfilé



L'état 3 est le premier à défiler et à développer

➤ Recherche guidée (A*)

Cette méthode de recherche est basée sur une fonction d'évaluation des nœuds comme information heuristique pour les ordonner dans ouverte, cette fonction est à valeurs réelles sur les nœuds. Nous rappelons la structure de cette fonction (vu dans le chapitre 1) :

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

Où :

$g(x)$ Représente le coût du chemin entre la racine et x

$h(x)$ Représente une estimation du coût restant entre x et un éventuel sommet solution (BUT) accessible à partir de x.

Dans notre projet, nous avons choisi 2 heuristiques h_1 et h_2 pour évaluer le coût qui reste pour atteindre le but : la **distance Euclidienne** et la **distance de Manhattan** respectivement.

| Distance Euclidienne | Distance de Manhattan |
|---|---|
| Pour un état du taquin, c'est la somme des distances de chaque case et sa position dans l'état but. | Pour un état du taquin, c'est la somme de nombre de cases qui séparent chaque case de cet état à sa position dans l'état but. |

Table 2.1- Définition des heuristiques

Remarque

Pour les trois méthodes de recherches citées ci-dessus, la recherche d'un nœud dans la liste ouverte ou fermée nécessite la sauvegarde des nœuds qui sont soit pas encore développés, c-à-d ils sont dans ouverte, soit déjà visitées et qui se trouvent dans fermée. Pour le jeu du taquin, le nombre d'états à stocker dans ouverte et fermée est important, et l'utilisation d'un simple tableau n'est pas le bon choix, car la recherche dans une grosse table est très lent (elle ce fait en parcourant tous le tableau jusqu'à trouver l'état recherché).

Pour bien réussir le procédé de la recherche, nous avons opté pour une structure de données assez puissante, c'est la **table de hachage**.

Dans notre projet, nous avons utilisé une **hashset**, une collection d'objets prédéfinie dans le langage java qui utilise une table de hachage pour le stockage. Les élé-

ments de cette table sont uniques, pour cette raison, nous avons utilisé une fonction qui retourne la représentation entière d'un état donné, et qui sera indexé dans cette table de façon unique comme suit :

$$M = \begin{bmatrix} 1 & 4 & 2 \\ 5 & 0 & 3 \\ 7 & 6 & 8 \end{bmatrix} \quad \xrightarrow{\text{Représentation entière}} \quad \begin{aligned} & 1 \times 10^8 + 4 \times 10^7 + 2 \times 10^6 + \\ & 5 \times 10^5 + 0 \times 10^4 + 3 \times 10^3 \\ & + 7 \times 10^2 + 6 \times 10^1 + 8 \times 10^0 \\ & = \textcolor{red}{142503768} \text{ (l'entier à indexer dans la table de hachage)} \end{aligned}$$

De cette manière, chaque état aura une représentation entière unique. Pour la recherche, nous avons associé une hashset pour la liste ouverte et une autre pour la liste fermée. Si un état est mis dans une de ces listes, la hashset correspondante ajoutera la représentation entière de cet état et qui sera ultérieurement utilisée pour la recherche dont le résultat sera trouvé en un temps $O(1)$.

2.2.2 Complexité théorique temporelle est spatiale

Les complexités temporelle est spatiale dépendent de :

b : facteur de branchement maximum de l'arbre de recherche.

d : profondeur à laquelle se trouve le meilleur nœud cible.

m : profondeur maximum de l'arbre de recherche.

Le tableau suivant résume la complexité théorique temporelle est spatiale de chaque méthode de recherche :

| Méthode | Complexité temporelle | Complexité temporelle |
|---------|-----------------------|-----------------------|
| DFS | $O(b^d)$ | $O(b^d)$ |
| BFS | $O(b^m)$ | $O(b*m)$ |
| A* | $O(b^d)$ | $O(b^d)$ |

Table 2.2- Complexité théorique temporelle et spatiale

Chapitre 3

Expérimentations

3.1 Environnement de travail

Afin de faire des tests sur notre jeu, nous avons utilisé une machine dont les caractéristiques sont montrées dans la figure 3.1.

Informations système générales

Édition Windows

Windows 10 Professionnel

© Microsoft Corporation. Tous droits réservés.

Système

Processeur : Intel(R) Core(TM) i5-6200U CPU @ 2.30GHz 2.40 GHz

Mémoire installée (RAM) : 8,00 Go (7,88 Go utilisable)

Type du système : Système d'exploitation 64 bits, processeur x64

Stylet et fonction tactile : La fonctionnalité d'entrée tactile ou avec un stylet n'est pas disponible sur cet écran.

Figure 3.1 - Performances de la machine utilisée

3.2 Outils utilisés

3.2.1 Langage de programmation

Nous avons choisi le langage **java** pour la résolution de notre problème qui nous a permis d'avoir une application bien structurée, modulable et efficace.

3.2.2 IDE

Pour l'environnement de travail, nous avons opté pour l'IDE **Eclipse** spécialement dédié au développement en utilisant le langage Java. Il est choisi pour sa simplicité de développement et sa meilleure gestion des projets.

3.3 Interface graphique

Pour rendre notre jeu du taquin facile à tester et bien visualiser le comportement des différentes méthodes implémentées, nous avons conçu une interface graphique simple à manipuler.

Cette interface est composée d'une fenêtre qui se décompose en 9 sections comme c'est montré dans la figure 3.2.

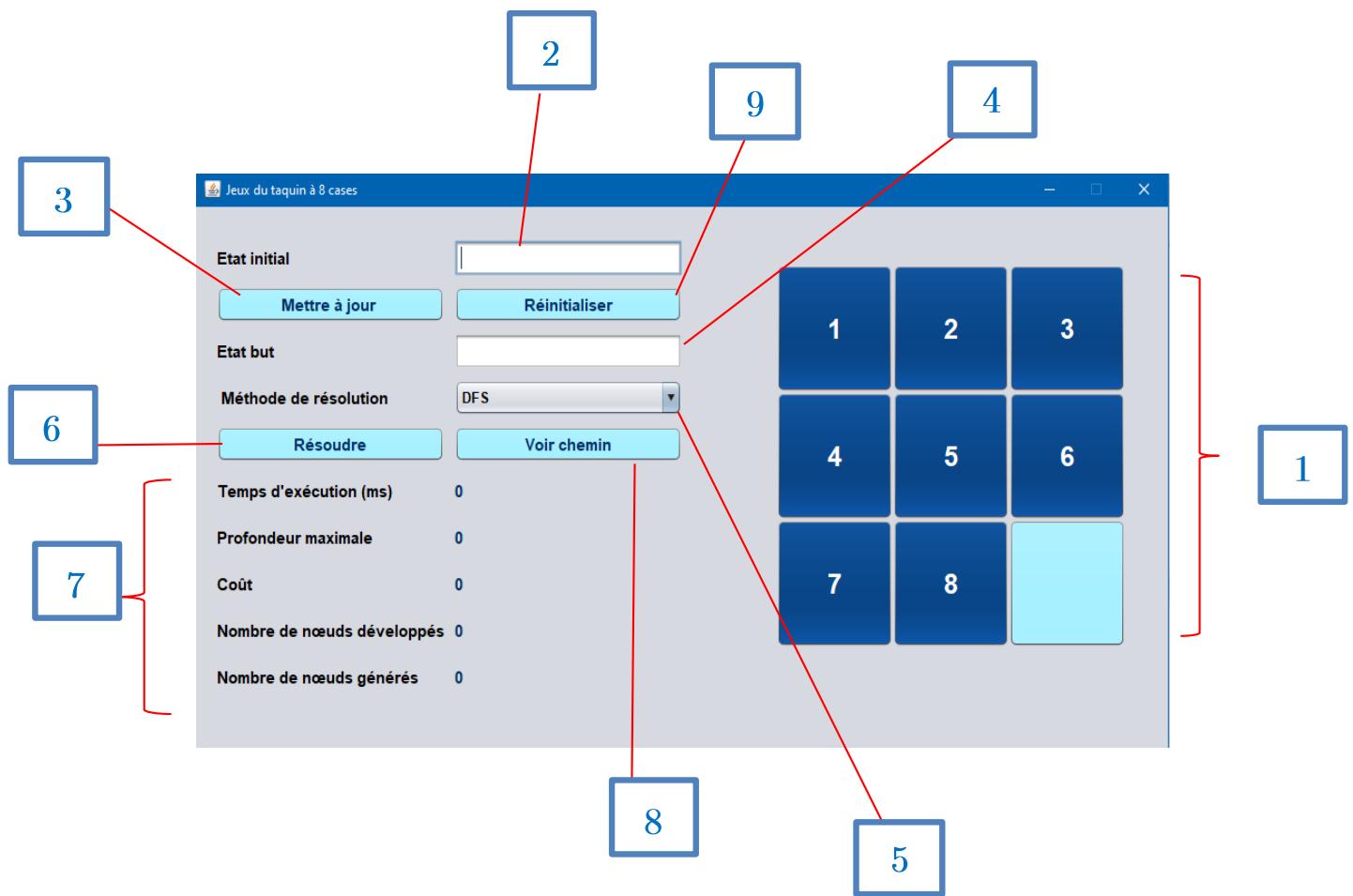


Figure 3.2 - Interface graphique du jeu de taquin

3.3.1 Détails

| Section | Description |
|---------|---|
| 1 | C'est une grille composée de 9 cases. Dans un premier temps, elle contient un exemple d'un état initial du taquin (default state), celui-ci change par la suite si l'utilisateur entre un nouvel état et met à jour la grille. |
| 2 | Une zone de texte qui permet à l'utilisateur d'enter l'état initial de l'instance du taquin qu'il a choisi. |
| 3 | En cliquant sur ce bouton, la grille se met à jour par le nouvel état initial entré dans la section 2. |
| 4 | Une zone de texte qui permet à l'utilisateur d'enter l'état but de l'instance du taquin qu'il a choisi. |
| 5 | Une barre de menu qui permet à l'utilisateur de choisir une méthode de résolution, soit : DFS, BFS, A*[Distance Euclidienne], A*[Distance de Manhattan]. |
| 6 | Ce bouton permet de résoudre l'instance de Taquin entrée par l'utilisateur avec la méthode sélectionnée. |
| 7 | Cette section affiche les différents résultats d'exécution dont l'utilisateur a besoin de visualiser afin d'évaluer la méthode de résolution choisie. Si l'instance n'admet aucune solution, le texte « Insoluble » est affiché sur ce bouton. |
| 8 | En cliquant sur ce bouton, l'utilisateur aura la possibilité de voir le chemin qui mène vers la solution obtenue. Chaque clique sur ce bouton montre un déplacement de la case vide (en haut, en bas, à gauche, à droite) jusqu'à arriver au but recherché. Une fois le but est atteint, le texte « Résolu » est affiché sur le bouton indiquant la fin du processus. |
| 9 | Ce bouton met à jour toutes les sections de l'interface graphique dans le cas où l'utilisateur souhaite tester une nouvelle instance du taquin. |

Table 3.1- Détails sur les différentes sections de l'interface

3.3.2 Jeu de test

Pour mettre en œuvre cette interface, nous avons testé une instance du taquin, les résultats sont montrés dans les figures suivantes :

- État initial

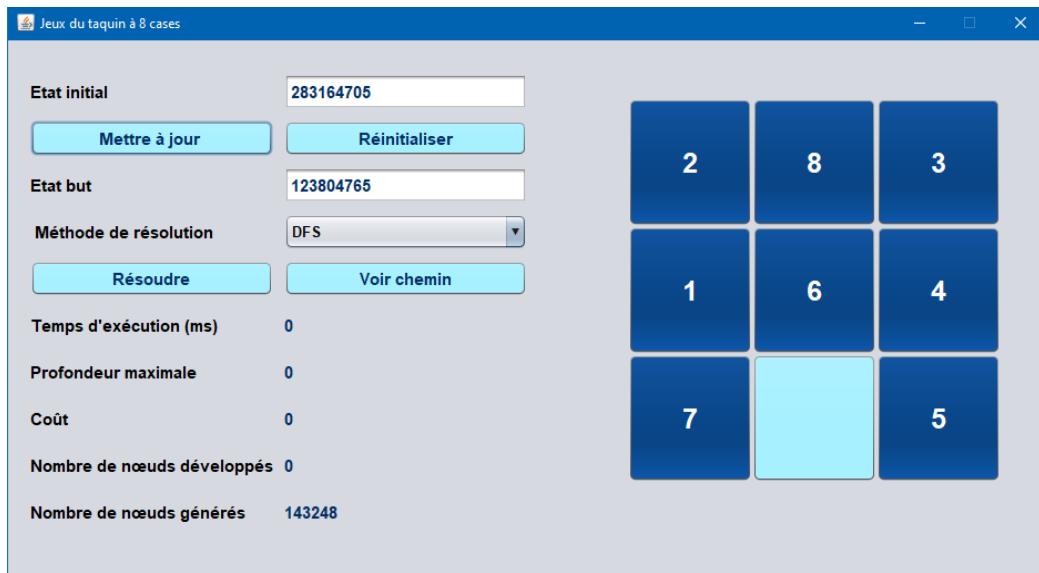


Figure 3.3 - Etat initial d'une instance de taquin

- Résolution par la méthode DFS

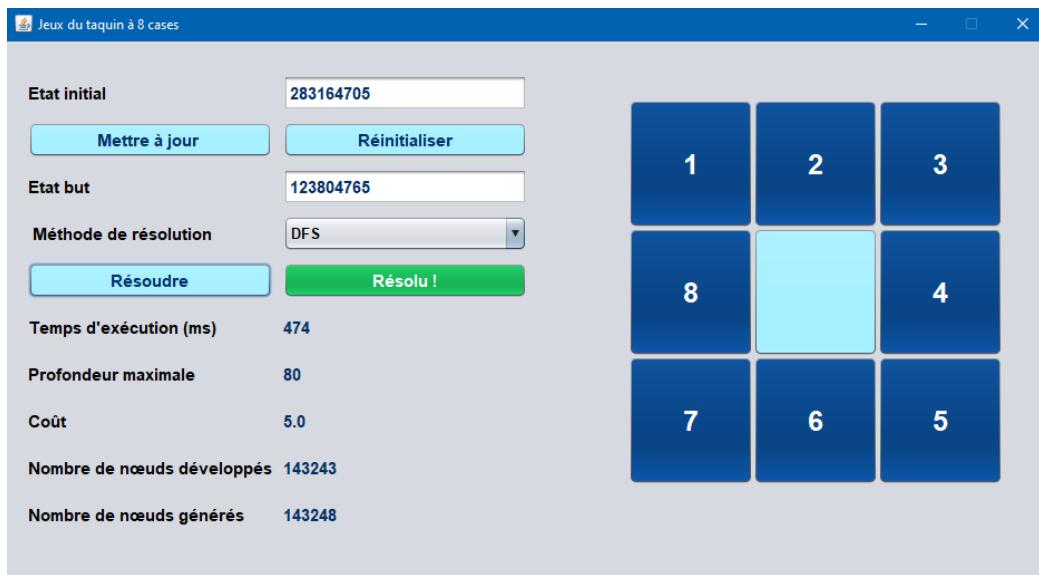


Figure 3.4 - Exécution (DFS)

- Résolution par la méthode BFS

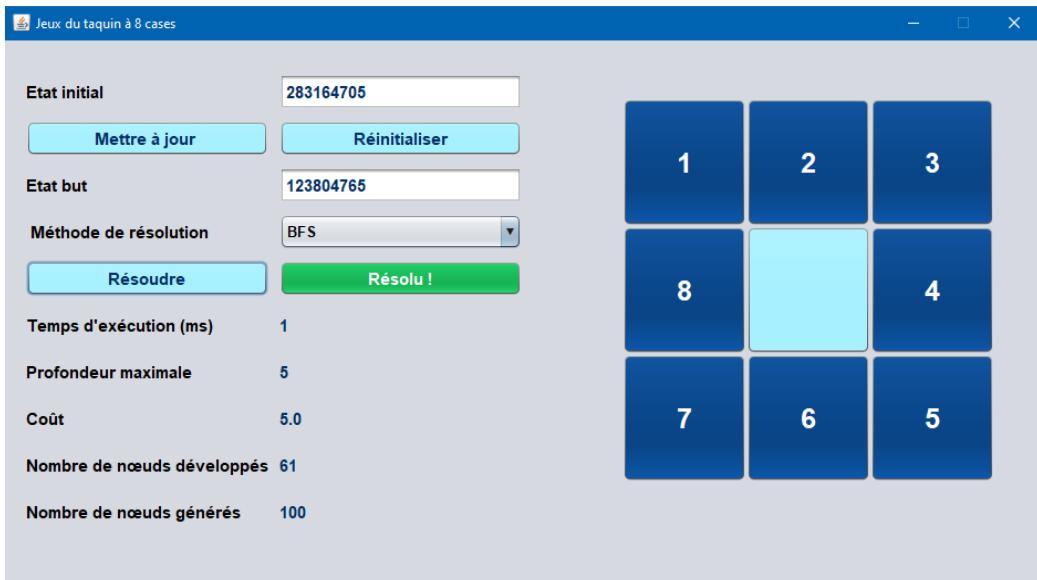


Figure 3.5 - Exécution (BFS)

- Résolution par la méthode A* (distance Euclidienne)

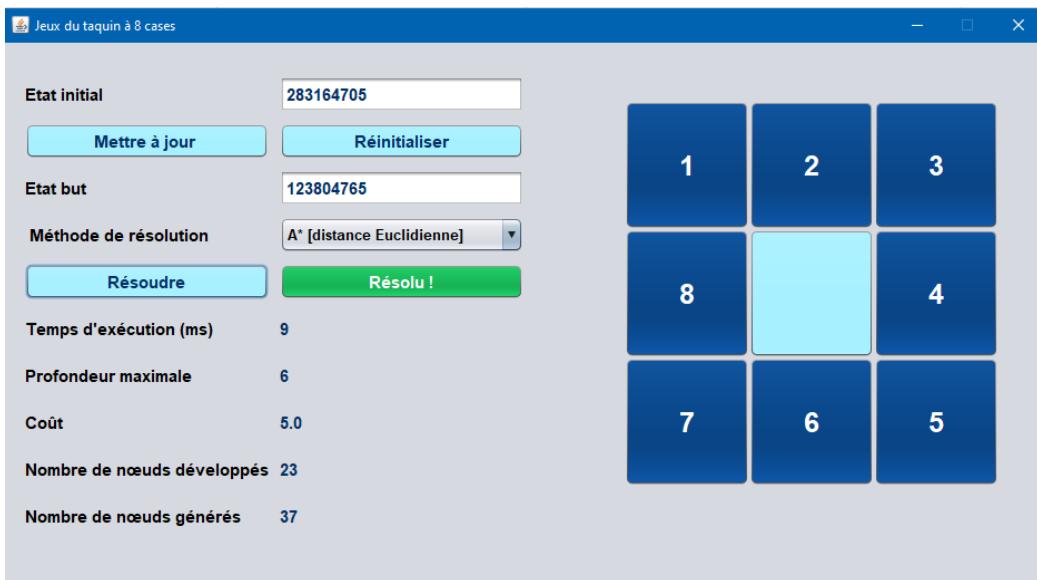


Figure 3.6 - Exécution (A*- distance Euclidienne)

- Résolution par la méthode A* (distance de Manhattan)

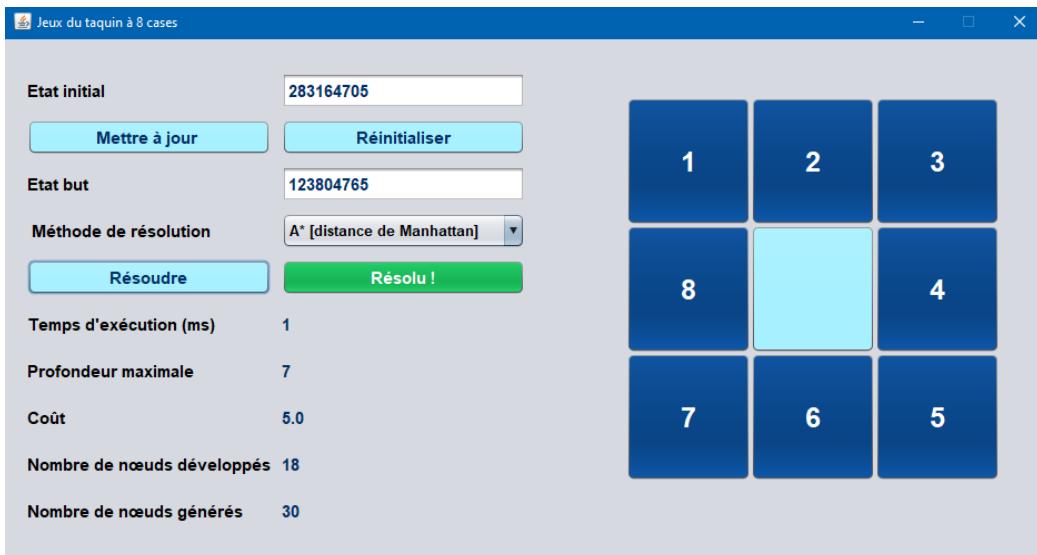


Figure 3.7 - Exécution (A*- distance de Manhattan)

- Instance insoluble

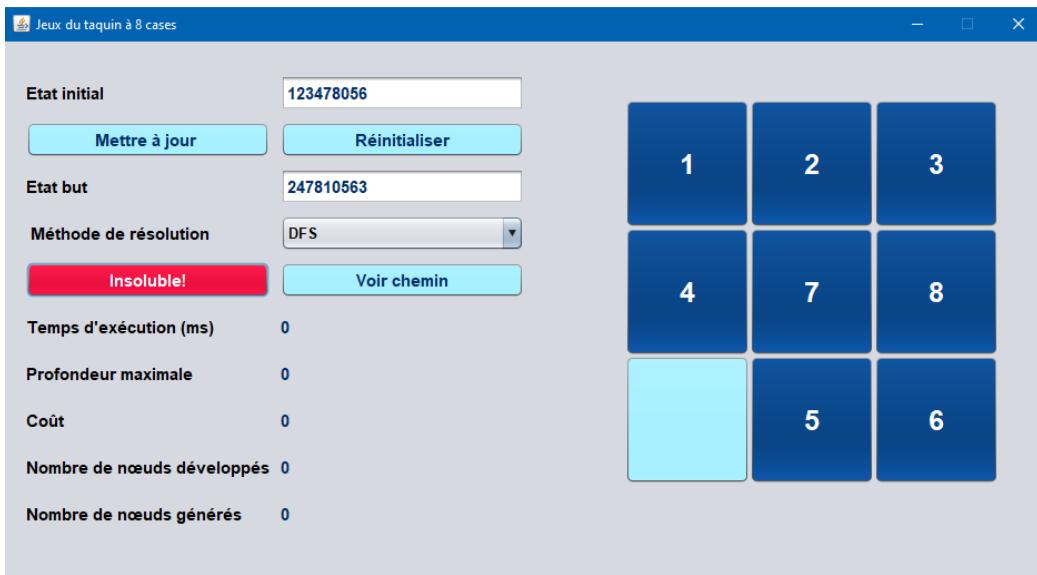


Figure 3.8 - Instance insoluble

3.4 Analyse, interprétation et comparaison des résultats

3.2.1 Instances

Afin de comparer les différentes méthodes de résolution, nous avons pris dix instances solvables du taquin, à savoir :

Instance 1 :

| | | |
|---|---|---|
| 2 | 8 | 3 |
| 1 | 6 | 4 |
| 7 | x | 5 |



| | | |
|---|---|---|
| 1 | 2 | 3 |
| 8 | x | 4 |
| 7 | 6 | 5 |

Instance 2 :

| | | |
|---|---|---|
| 2 | 5 | 8 |
| x | 1 | 3 |
| 4 | 7 | 6 |



| | | |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 8 | x |

Instance 3 :

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 8 | x |



| | | |
|---|---|---|
| x | 1 | 2 |
| 3 | 4 | 5 |
| 6 | 7 | 8 |

Instance 4 :

| | | |
|---|---|---|
| 7 | 2 | 4 |
| 5 | x | 6 |
| 8 | 3 | 1 |



| | | |
|---|---|---|
| x | 1 | 2 |
| 3 | 4 | 5 |
| 6 | 7 | 8 |

Instance 5 :

| | | |
|---|---|---|
| 1 | 4 | 2 |
| 5 | x | 3 |
| 7 | 6 | 8 |



| | | |
|---|---|---|
| x | 1 | 2 |
| 3 | 4 | 5 |
| 6 | 7 | 8 |

Instance 6 :



| | | |
|---|---|----------|
| 1 | 2 | 3 |
| 8 | 4 | 7 |
| 6 | 5 | x |

| | | |
|---|----------|---|
| 1 | 2 | 3 |
| 8 | x | 4 |
| 7 | 6 | 5 |

Instance 7 :



| | | |
|---|---|----------|
| 8 | 7 | 6 |
| 5 | 4 | 3 |
| 2 | 1 | x |

| | | |
|----------|---|---|
| x | 1 | 2 |
| 3 | 4 | 5 |
| 6 | 7 | 8 |

Instance 8 :



| | | |
|----------|---|---|
| 1 | 2 | 3 |
| x | 4 | 6 |
| 7 | 5 | 8 |

| | | |
|---|---|----------|
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 8 | x |

Instance 9 :



| | | |
|---|---|----------|
| 1 | 2 | 3 |
| 5 | 6 | x |
| 7 | 8 | 4 |

| | | |
|----------|---|---|
| 1 | 2 | 3 |
| 5 | 8 | 6 |
| x | 7 | 4 |

Instance 10 :



| | | |
|---|----------|---|
| 1 | x | 3 |
| 2 | 4 | 6 |
| 5 | 7 | 8 |

| | | |
|----------|---|---|
| x | 1 | 2 |
| 3 | 4 | 5 |
| 6 | 7 | 8 |

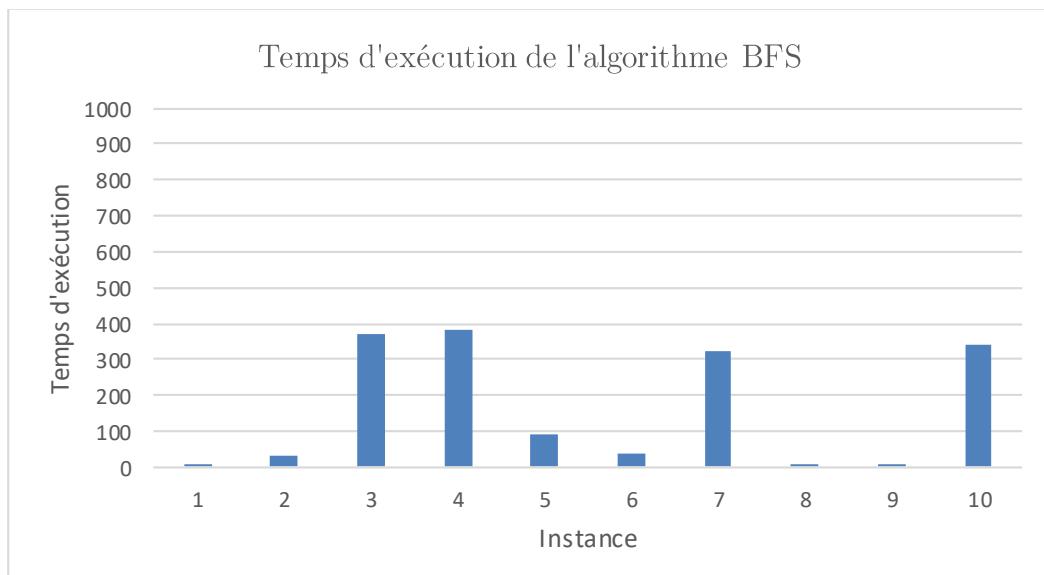
3.2.2 Résultats

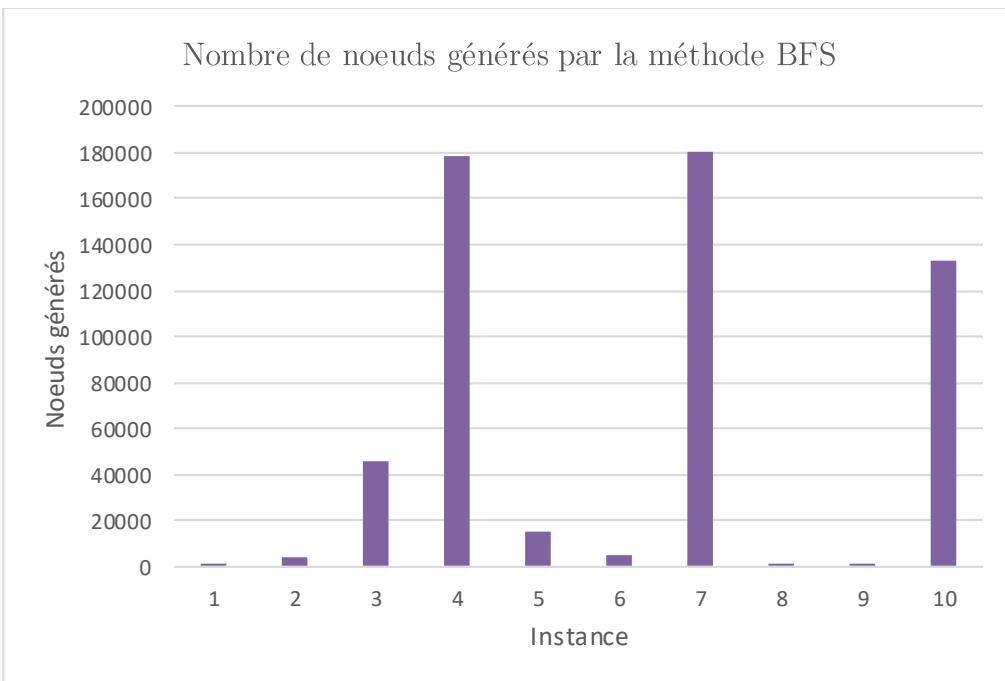
Nous avons lancé l'exécution des 10 instances du taquin décrites ci-dessus pour chaque méthodes de résolution (BFS, DFS, A*). Les résultats sont les suivants :

1. En largeur d'abord (BFS)

| Instance | Temps d'exécution | Profondeur MAX | Coût | Nœuds développés | Nœuds générés |
|----------|-------------------|----------------|------|------------------|---------------|
| 1 | 1 | 5 | 5 | 61 | 100 |
| 2 | 33 | 13 | 13 | 2615 | 4164 |
| 3 | 372 | 22 | 22 | 80620 | 45368 |
| 4 | 384 | 26 | 26 | 172835 | 178116 |
| 5 | 93 | 16 | 16 | 9583 | 15194 |
| 6 | 38 | 14 | 14 | 2903 | 4806 |
| 7 | 325 | 28 | 28 | 178036 | 180854 |
| 8 | 1 | 3 | 3 | 13 | 22 |
| 9 | 1 | 3 | 3 | 14 | 25 |
| 10 | 344 | 23 | 23 | 110052 | 132905 |

Table 3.2- Résultats d'exécution en largeur d'abord



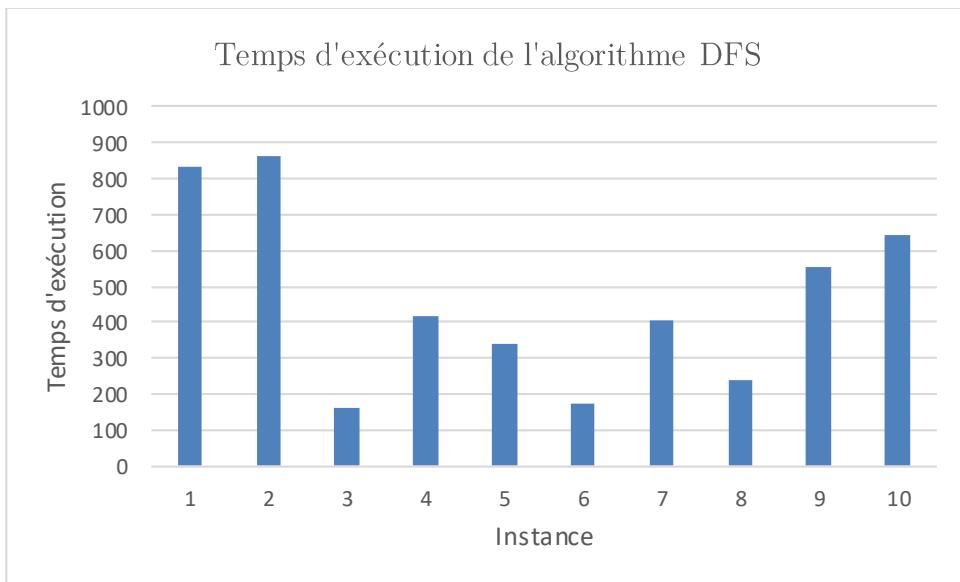


2. En profondeur d'abord (DFS)

Le tableau suivant représente les résultats d'exécution de l'algorithme DFS qui résout le problème du taquin sans aucune limite de profondeur :

| Ins-tance | Temps d'exécution | Profon-deur MAX | Coût | Nœuds déve-loppés | Nœuds gé-nérés |
|-----------|-------------------|-----------------|--------|-------------------|----------------|
| 1 | 834 | 66997 | 5 | 181435 | 181441 |
| 2 | 865 | 66743 | 19665 | 165536 | 180563 |
| 3 | 160 | 21744 | 21744 | 22708 | 128442 |
| 4 | 417 | 59626 | 596269 | 72230 | 111770 |
| 5 | 341 | 45342 | 45342 | 50127 | 81971 |
| 6 | 174 | 35388 | 35388 | 37799 | 63330 |
| 7 | 404 | 67100 | 65676 | 108205 | 150323 |
| 8 | 241 | 61578 | 61577 | 76424 | 116984 |
| 9 | 555 | 67099 | 12285 | 171708 | 181172 |
| 10 | 646 | 67471 | 31843 | 155499 | 178680 |

Table 3.3- Résultats d'exécution en profondeur d'abord



Maintenant, nous allons tester cette méthode avec différents seuils de profondeur et enregistrons à chaque fois le nombre de nœuds générés ainsi que le nombre de nœuds développés. Nous avons fixé 3 seuils de profondeur dont les résultats sont les suivants :

➤ Seuil = 50

| Instance | Nœuds développés | Nœuds générés |
|----------|------------------|---------------|
| 1 | 10446 | 10452 |
| 2 | 34337 | 34353 |
| 3 | 65822 | 65844 |
| 4 | 10284 | 10286 |
| 5 | 41689 | 41702 |
| 6 | 17480 | 17502 |
| 7 | 56766 | 56792 |
| 8 | 31429 | 31435 |
| 9 | 7132 | 7160 |
| 10 | 1122 | 1130 |

Table 3.4- Résultats d'exécution en profondeur d'abord (seuil = 50)

➤ Seuil = 80

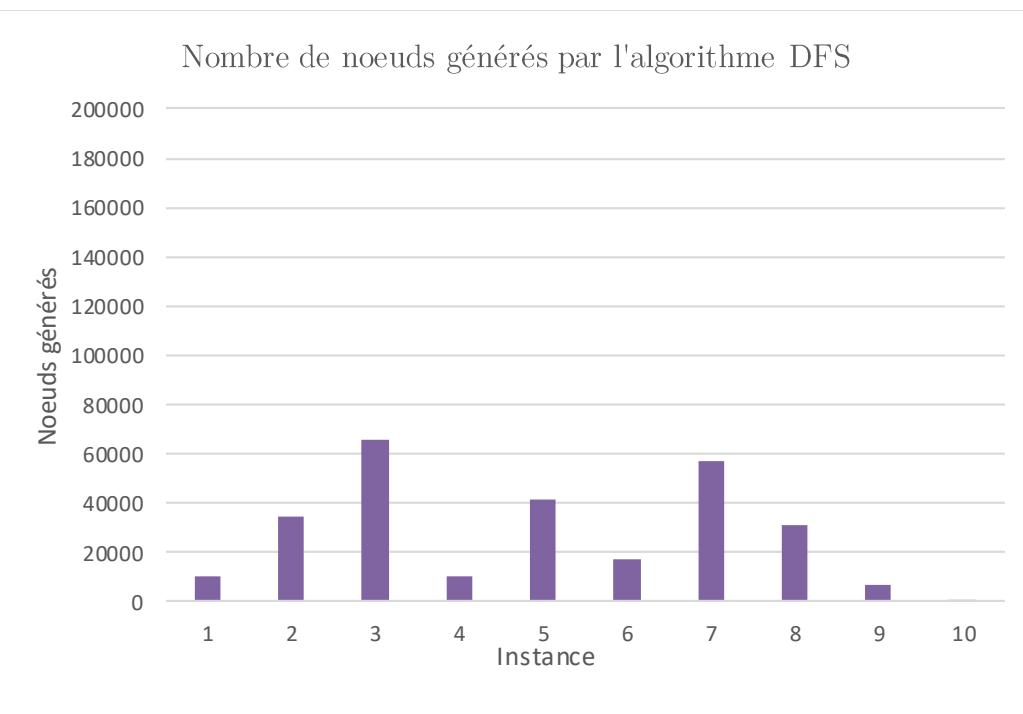
| Instance | Nœuds développés | Nœuds générés |
|----------|------------------|---------------|
| 1 | 14243 | 14243 |
| 2 | 50849 | 50900 |
| 3 | 13230 | 13269 |
| 4 | 12869 | 12812 |
| 5 | 15201 | 15205 |
| 6 | 8719 | 8779 |
| 7 | 11632 | 11642 |
| 8 | 93886 | 93921 |
| 9 | 11873 | 11899 |
| 10 | 57659 | 57774 |

Table 3.5- Résultats d'exécution en profondeur d'abord (seuil = 80)

➤ Seuil = 100

| Instance | Nœuds développés | Nœuds générés |
|----------|------------------|---------------|
| 1 | 147942 | 147942 |
| 2 | 141619 | 141635 |
| 3 | 80620 | 103699 |
| 4 | 40138 | 40210 |
| 5 | 26192 | 26269 |
| 6 | 43858 | 43920 |
| 7 | 46075 | 46138 |
| 8 | 78544 | 78560 |
| 9 | 14506 | 14571 |
| 10 | 10035 | 10080 |

Table 3.6- Résultats d'exécution en profondeur d'abord (seuil = 100)

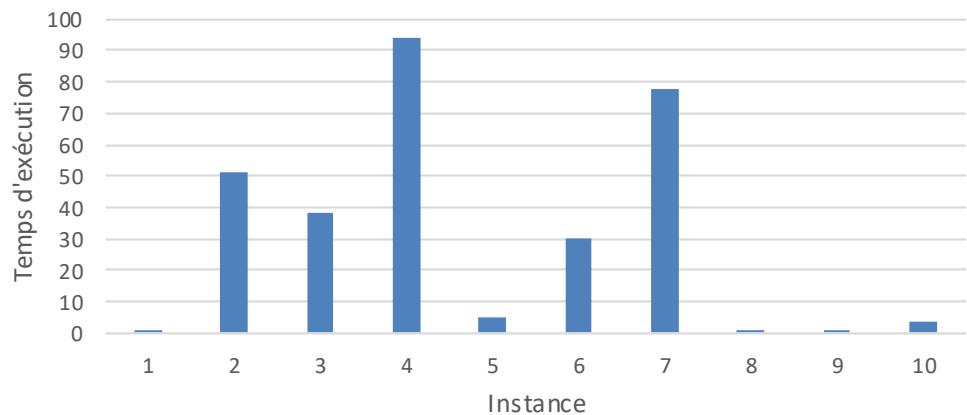


3. A* avec l'heuristique « Distance Euclidienne »

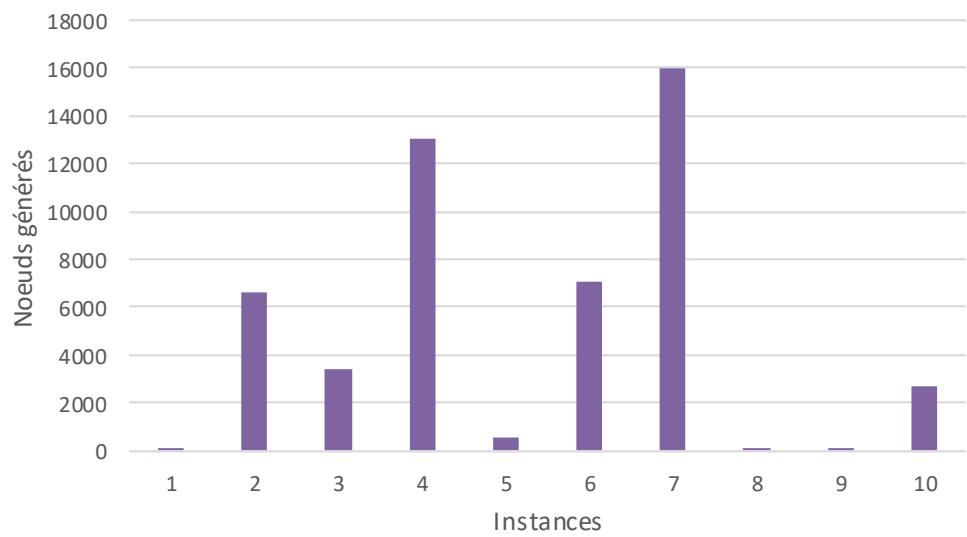
| Inst- ance | Temps d'exécu- tion | Profon- deur MAX | Coût | Nœuds dé- veloppés | Nœuds géné- rés |
|---------------|---------------------------|------------------------|------|-----------------------|--------------------|
| 1 | 1 | 6 | 5 | 23 | 37 |
| 2 | 51 | 22 | 13 | 4415 | 6639 |
| 3 | 38 | 22 | 22 | 2161 | 3440 |
| 4 | 94 | 26 | 26 | 8845 | 13036 |
| 5 | 5 | 16 | 16 | 337 | 555 |
| 6 | 30 | 22 | 14 | 4533 | 7096 |
| 7 | 78 | 28 | 28 | 10381 | 16021 |
| 8 | 1 | 8 | 3 | 41 | 72 |
| 9 | 1 | 9 | 3 | 53 | 91 |
| 10 | 4 | 23 | 23 | 1676 | 2649 |

Table 3.7- Résultats d'exécution de l'algorithme A*(Distance Euclidienne)

Temps d'exécution de la méthode A* avec l'heuristique "distance Euclidienne"



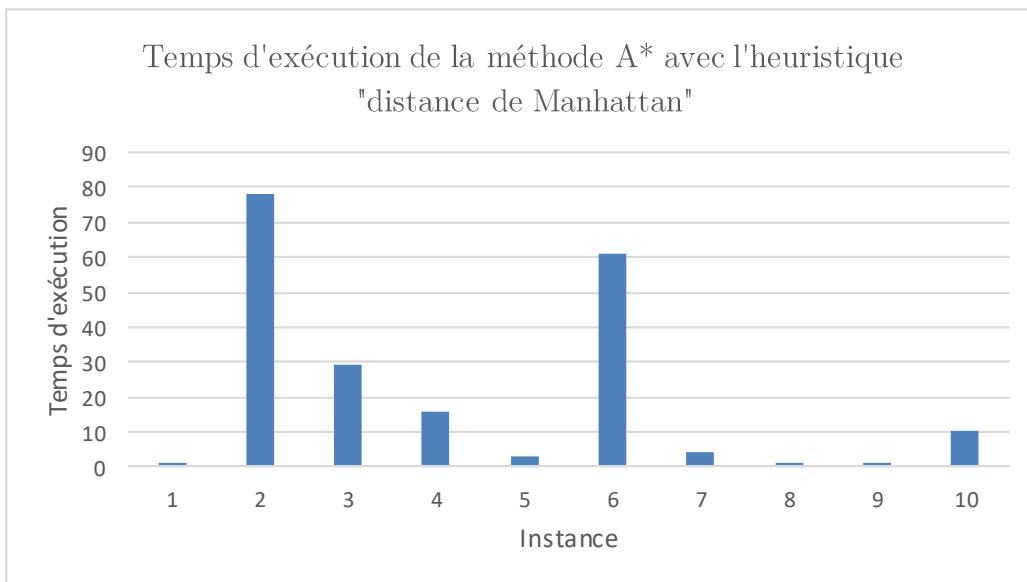
Nombre de noeuds générés par la méthode A* avec l'heuristique "distance Euclidienne"

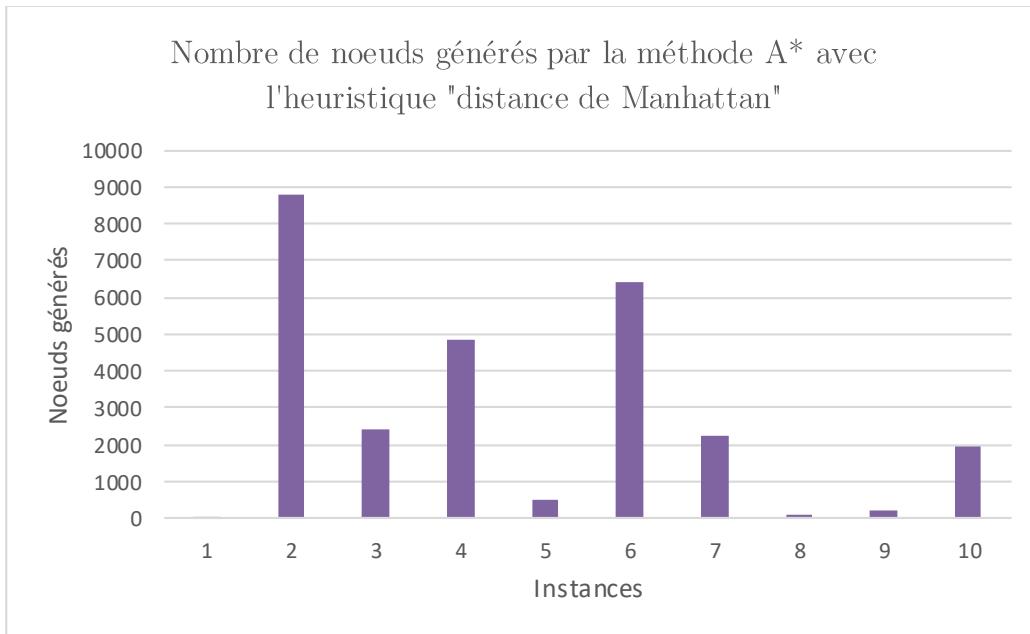


4. A* avec l'heuristique « Distance de Manhattan »

| Instance | Temps d'exécution | Profondeur MAX | Coût | Nœuds développés | Nœuds générés |
|----------|-------------------|----------------|------|------------------|---------------|
| 1 | 1 | 7 | 5 | 18 | 30 |
| 2 | 78 | 25 | 15 | 5802 | 8762 |
| 3 | 29 | 22 | 22 | 1516 | 2419 |
| 4 | 16 | 26 | 26 | 3093 | 4829 |
| 5 | 3 | 16 | 16 | 344 | 511 |
| 6 | 61 | 23 | 14 | 4139 | 6416 |
| 7 | 4 | 28 | 28 | 1427 | 2212 |
| 8 | 1 | 12 | 3 | 39 | 66 |
| 9 | 1 | 14 | 3 | 142 | 233 |
| 10 | 10 | 23 | 23 | 1239 | 1952 |

Table 3.8- Résultats d'exécution de l'algorithme A*(Distance de Manhattan)



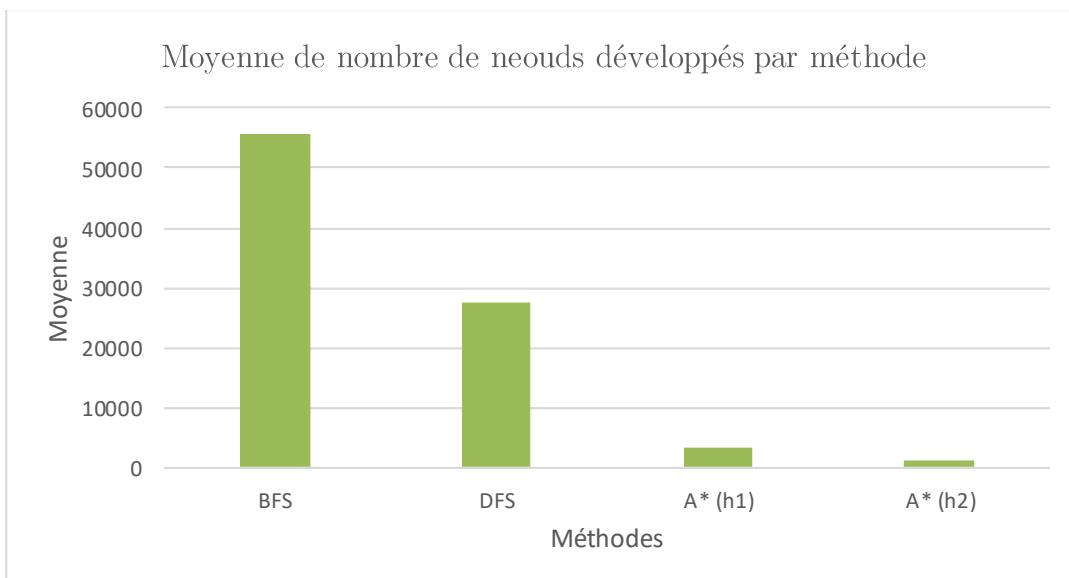
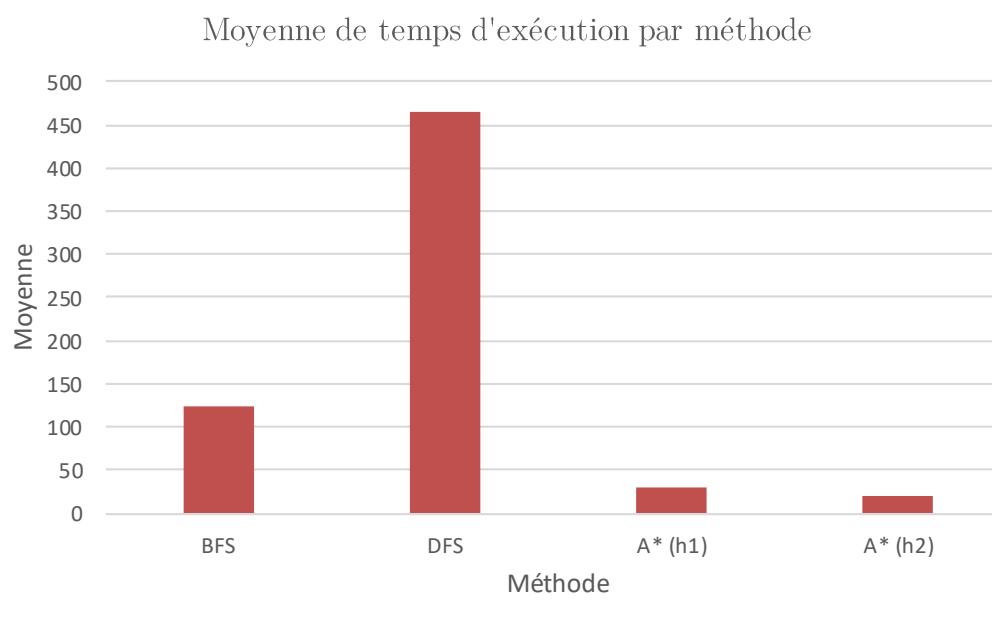


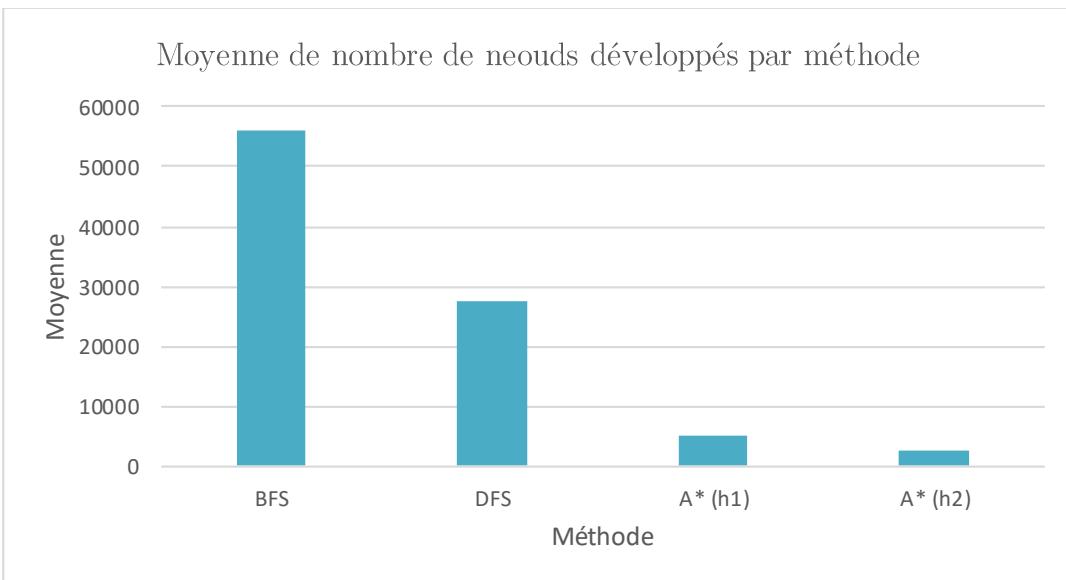
3.5 Statistiques

Après avoir obtenu et représenté les résultats d'exécution de chaque méthode de recherche, nous allons, dans cette partie, récapituler ces derniers dans un tableau afin d'obtenir les statistiques de cette étude. Puis les représenter dans des graphes comparatifs.

| Méthodes | BFS | DFS | A*(h1) | A* (h2) |
|--|-------|-------|--------|---------|
| Moyenne de temps d'exécution (ms) | 123 | 464 | 30 | 20 |
| Moyenne de nombre de noeuds développés | 55673 | 27650 | 3247 | 1775 |
| Moyenne de nombre de noeuds générés | 56155 | 27665 | 4964 | 2743 |

Table 3.9- statistiques





3.6 Étude comparative

Nous allons passer en revue une comparaison des différentes méthodes de résolution du problème du taquin étudiées dans ce projet, en se basant sur la rapidité d'exécution (temps d'exécution ou nombre de noeuds développés) et l'espace mémoire consommé (nombre de noeuds générés).

Après observation et analyse des résultats obtenus des tests des trois algorithmes de résolutions sur plusieurs instances du taquin, nous avons remarqué clairement que les deux méthodes de recherche en largeur d'abord et en profondeur d'abord sont des méthodes simples et exactes, mais qui ont des limites en terme de temps d'exécution et espace mémoire utilisé. En effet, la méthode de recherche en largeur d'abord s'exécute en un temps abordable par rapport à la recherche en profondeur d'abord qui est complexe en temps de recherche, pour un espace d'états de grande taille, il est très difficile d'atteindre un but. Par contre, cette méthode n'est pas gourmande en mémoire par rapport à la méthode en largeur d'abord qui génère un nombre de noeud très important et qui amène vers une explosion combinatoire très rapidement surtout avec des instances de grandes tailles.

Il est bien remarquable que l'ajout d'une heuristique (l'algorithme A* avec les 2 heuristiques h1 et h2) a amélioré le processus de recherche en réduisant le temps d'exécution d'une façon significative (instances résolues en quelques secondes). De

plus, l'espace mémoire consommé par cette méthode est souvent abordable. Cependant, cette méthode souffre de quelques lacunes à ne pas négliger, qui étant la difficulté de trouver de bonnes heuristiques admissibles et par conséquent, la difficulté de trouver la solution optimale (qui est dans notre cas le chemin le moins couteux qui mène vers un état cible).

Conclusion

En dépit de la simplicité du problème du taquin, il est très difficile à résoudre avec les méthodes classiques. Ce qui nous oblige à trouver d'autres méthodes aussi performantes. Nous pouvons constater que les méthodes guidées sont beaucoup plus utiles que les méthodes aveugles.

Il faut considérer chaque situation, le type et la quantité des données, les moyens et le temps à notre disposition pour choisir les techniques et les heuristiques à appliquer. Souvent le monde de l'industrie préfère une réponse approximative mais rapide et efficace plutôt que l'utopique réponse parfaite.

Bibliographie

- [1] : Agrégation externe de mathématiques, session 2008 Épreuve de modélisation, option informatique
- [2] : Henri Farreny, Article : « Des heuristiques plus performantes si avec la fonction on garde la raison » ENSEEIHT-INP de Toulouse / Institut de Recherches en Informatique de Toulouse. <http://archive.stock.free.fr/mémoire/Farreny.pdf>
- [3] : Drias, H. Z. Algorithmique Moderne Analyse et Complexité. Alger : Office des Publications Universitaires, 2017.
- [4] : Maria Malek - Grande École de sciences, d'ingénierie, d'économie et de gestion, -Cours de Deuxième année : Recherche dans un espace d'états.
<http://mma.perso.eisti.fr/HTML-IA/Cours/Cours1/ch1.pdf>
- [5] : Pr. Azzoune Hamid, Université des Sciences et de la Technologie Houari Boumedién - Support de cours Résolution des problèmes master 1 SII : Stratégies de Recherche pour les SP.