**République Algérienne Démocratique et Populaire**

**Ministère de l’Enseignement Supérieur et de la Recherche Scientifique**

**Université des Sciences et de la Technologie Houari Boumediene**

**Faculté d’Informatique**

Département d’Intelligence Artificielle et Sciences des Données

Mini-projet Partie n°1

**Méta-heuristiques et algorithmes évolutionnaires**

**Groupe 15**:

**BOULARIACHE Abdessamed, G3, 171731033493.**

**KEMOUM Meroua, G3, 171731053329**  **Professeurs:**

**MAHIDDINE Mohamed Amine, G3, 201704000012.** Mme. Drias

Mme. Moulai

**TAZIR Mohamed Reda, G3, 161631076578.**

Date 09/04/2022

**Table des matières**

[Introduction générale 4](#_Toc100717558)

[Etude théorique du problème de taquin 4](#_Toc100717559)

[1. Définition formelle du problème de taquin: 5](#_Toc100717560)

[2. La présentation de La modélisation de la solution: 6](#_Toc100717561)

[2.1 Décrire la structure des états (configurations) du problème. 6](#_Toc100717562)

[2.2 Les règles de changements d’états: 7](#_Toc100717563)

[2.3 La solution : 7](#_Toc100717564)

[3. La présentation des algorithmes de résolution et la Complexité théorique: 7](#_Toc100717565)

[3.1 Algorithme de résolution en largeur d’abord ou BFS: 8](#_Toc100717566)

[3.1.1 Explication de fonctionnement de l'algorithme: 8](#_Toc100717567)

[3.1.2 Le pseudocode de l'algorithme: 8](#_Toc100717568)

[3.1.3 Exemple de résolution: 9](#_Toc100717569)

[3.1.4 Etude de Complexité : 9](#_Toc100717570)

[3.2 Algorithme de résolution en profondeur d’abord ou DFS: 10](#_Toc100717571)

[3.2.1 Explication de fonctionnement de l'algorithme: 10](#_Toc100717572)

[3.2.2 Le pseudocode de l'algorithme: 10](#_Toc100717573)

[3.2.3 Exemple de résolution: 11](#_Toc100717574)

[3.2.4 Complexité : 12](#_Toc100717575)

[3.3 Algorithme de résolution avec A\* 12](#_Toc100717576)

[3.3.1 Explication de fonctionnement de l'algorithme: 12](#_Toc100717577)

[3.3.2 Le pseudocode de l'algorithme A\*: 12](#_Toc100717578)

[3.3.3 Exemple de résolution (case mal placé): 14](#_Toc100717579)

[3.3.4 Exemple de résolution (Manhattan): 15](#_Toc100717580)

[3.3.5 Complexité : 15](#_Toc100717581)

[4. Modélisation et structures de données : 17](#_Toc100717582)

[4.1. Représentation d’un état : 17](#_Toc100717583)

[4.2. Génération de l’état initial: 17](#_Toc100717584)

[Test de solvabilité : 17](#_Toc100717585)

[4.3. Exploration des états: 18](#_Toc100717586)

[4.4. Fonction de successeur: 19](#_Toc100717587)

[4.5. Actions : 19](#_Toc100717588)

[5. Présentation de l’Interface graphique 19](#_Toc100717589)

[6. Etude Expérimentale: 22](#_Toc100717590)

[6.1. Temps d'exécution (en millisecondes): 22](#_Toc100717591)

[6.2 Nombre de nœuds développés: 22](#_Toc100717592)

[6.3. La profondeur : 23](#_Toc100717593)

[6.3. Graphes: 24](#_Toc100717594)

[6.2 Comparaison entre méthode : 27](#_Toc100717595)

[Conclusion 27](#_Toc100717596)

**Table des figures**

[**Figure 1** Un exemple d’un problème de taquin avec 4\*4 cases. 5](#_Toc100715712)

[**Figure 2 :** Une configuration initiale et la configuration cible d’un problème de taquin avec 4\*4 cases. 6](#_Toc100715713)

[**Figure 3:** La représentation de l'état initial et l'état but de problème de taquin. 6](#_Toc100715714)

[**Figure 4 :** Algorithme de résolution en largeur d’abord. 8](#_Toc100715715)

[**Figure 5**: Exemple de résolution de l’algorithme de recherche en largeur d’abord. 9](#_Toc100715716)

[**Figure 6:** Exemple de résolution de l’algorithme de recherche en largeur d’abord cas de problème de taquin. 9](#_Toc100715717)

[**Figure 7:**  Algorithme de résolution en profondeur d’abord. 11](#_Toc100715718)

[**Figure 8:**  Exemple de résolution de l’algorithme de recherche en profondeur d’abord. 11](#_Toc100715719)

[**Figure 9:** Algorithme de résolution avec A\* 13](#_Toc100715720)

[**Figure 10 :** Exemple de résolution de l’algorithme de recherche A\* h1 (case mal placé) 14](#_Toc100715721)

[**Figure 11 :** Exemple de résolution de l’algorithme de recherche A\* h2 (Manhattan) 15](#_Toc100715722)

[**Figure 12 :** Comparaison entre les différents algorithmes 16](#_Toc100715723)

[**Figure 14 :** Algorithme de la fonction de hachage 18](#_Toc100715724)

[**Figure 15 :** Page d’accueil 19](#_Toc100715725)

[**Figure 16 :** Page principale 20](#_Toc100715726)

[**Figure 17 :** Méthodes de résolution 20](#_Toc100715727)

[**Figure 18 :** Niveaux de difficulté 20](#_Toc100715728)

[**Figure 19 :** Page montrant les résultats après résolution 21](#_Toc100715729)

[**Figure 20 :** Visualisation du déroulement de la recherche 21](#_Toc100715730)

[**Figure 21 :** graphique comparatif du temps d'exécution obtenu par les quatre méthodes de résolution 24](#_Toc100715731)

[**Figure 22 :** graphique comparatif du nombre de nœuds développés par les quatre méthodes de résolution 25](#_Toc100715732)

[**Figure 23 :** graphique comparatif de la profondeur atteinte par les quatre méthodes de résolution 26](#_Toc100715733)

# **Liste des tableaux**

[**Tableau 1 :** Les règles de changements d’états pour un taquin de n\*n cases. 7](#_Toc100715920)

[**Tableau 2**: Le temps d'exécution obtenus par les quatre méthodes de résolution 22](#_Toc100715921)

[**Tableau 3 :** Le nombre de nœuds développés par les quatre méthodes de résolution 23](#_Toc100715922)

[**Tableau 4** : La profondeur atteinte par les quatre méthodes de résolution 23](#_Toc100715923)

# Introduction générale

En théorie de la complexité, un problème **NP-complet** est un problème de décision vérifiant deux propriétés qui sont: la possibilité de vérification efficace d’une solution (en temps polynomial) ou la classe des problèmes vérifiant cette propriété est notée NP. La seconde est que tous les problèmes de la classe NP se ramènent à celui-ci via une réduction polynomiale, qui signifie que le problème est au moins aussi difficile que tous les autres problèmes de la classe NP. La classe des problèmes NP-complets englobe les problèmes de décision les plus difficiles de la classe NP et sa définition s'appuie sur la notion de transformation polynomiale entre problèmes **[1]**.

L’utilisation de problèmes de jeu est très efficace pour avoir un premier pas dans le domaine de résolution de problème, au contraire des problèmes de monde réel qui nécessite des années d'expérience pour les maîtriser à cause de leur nature très complexe. Dans ce mini projet, on est concerné par le problème du taquin. Trouver le plus court chemin entre deux configurations du Taquin est considéré NP-hard. Le but de cette première partie de mini projet est **l’étude** et **le développement** des différentes méthodes **aveugles** (BFS, DFS) et **heuristique** (A\* avec deux heuristiques différentes) afin de résoudre le problème de taquin, faire une étude détaillée des différentes méthodes, la présentation de solution sous forme d’interface graphique développée.

# Etude théorique du problème de taquin

Le **problème de taquin** ou **15-puzzle** est un problème bien connu en Intelligence Artificielle, et particulièrement en résolutions de problème. Il a une longue histoire qui remonte aux années 1870. Ce dernier est composé de 15 carreaux numérotés de 1 à 15 qui glissent dans un cadre prévu pour 16, la **figure 1** représente une configuration possible.

Le but du jeu est d’organiser ce shuﬄedset**1** de 15 carreaux dans l’ordre croissant à partir d'un état initial quelconque avec la case vide qui est dans la dernière case.

1- Shuﬄedset : un plateau carré subdivisé en n\*n carreaux ou on peut les glisser vers un espace vide sans les prendre en main.

|  |
| --- |
|  |

**Figure 1 :** Un exemple d’un problème de taquin avec 4\*4 cases.

**Le jeu de taquin** appartient à la catégorie des problèmes d’énoncés de type *états* / *opérateurs de changement d'état*. Un tel énoncé contient un état **initial**, des états **finaux** à atteindre ou leurs caractérisations et des opérateurs <conditions, effets>. Il s'agit alors de découvrir un enchaînement d'opérateurs permettant de passer de l'état initial à un état final.

Propriétés d'une recherche:

**Terminaison :** si une solution existe, la recherche s'arrête en fournissant une solution. **Admissibilité :** la recherche fournit systématiquement la meilleure solution.

**Complexité :** le coût de la recherche, en particulier la taille du graphe de recherche par rapport au graphe de résolution.

Le problème de jeu de taquin est reconnu comme un problème difficile à résoudre et qui présentait un défi pour les mathématiciens et informaticiens de cette époque.

## Définition formelle du problème de taquin:

Le jeu du Taquin est simple à modéliser en termes d'états et de transitions entre états, mais difficile à résoudre du fait du nombre prohibitif d'états possibles. C'est un jeu de patience qui consiste en un damier carré de dimension n x n contenant n² - 1 chiffres compris entre 1 et (n² -l) et une case vide. La seule opération autorisée consiste à déplacer un chiffre adjacent à la case vide. Le jeu consiste à effectuer une série de déplacements autorisés à partir d'une configuration initiale pour aboutir à une configuration cible énoncée au début du jeu. **[1]**

|  |
| --- |
|  |

**Figure 2 :** Une configuration initiale et la configuration cible d’un problème de taquin avec 4\*4 cases.

## **La présentation de La modélisation de la solution:**

Pour résoudre ce problème, il faut spécifier ses 3 composantes:

### **2.1 Décrire la structure des états (configurations) du problème.**

La représentation informatique d’un état de ce problème peut être **une matrice n\*n**. L'ensemble des états constitue ce qu’on appelle l’espace du problème (pour un exemple de 3\*3 le nombre d’états est de l’ordre de 9! =362880 qui est relativement petite). Pour un Taquin 5x5 ce nombre est de l'ordre 1025 (Le nombre de configurations possibles étant égal à **(n²)!**).

Example:

Pour un taquin de 9 cases. La représentation est définie comme la **figure 3**:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| | **Configuration initial** | | |  |  | **Configuration cible** | | | | --- | --- | --- | --- | --- | --- | --- | --- | | **1** | **7** | **4** |  |  | **1** | **2** | **3** | | **6** | **5** | **3** |  | **=>** | **8** |  | **4** | |  | **2** | **8** |  |  | **7** | **6** | **5** | |

**Figure 3:** La représentation de l'état initial et l'état but de problème de taquin.

* **L’état initial** toute configuration peut être désignée comme état initial, dans notre implémentation nous avons défini une fonction qui peut générer cet état selon le niveau de difficulté choisi (Easy, Medium, Hard).
* **l’état but** L’état but est unique et fixé au début du jeu par une fonction.

### **2.2 Les règles de changements d’états:**

**Les règles de changements d’états** sont les déplacements de la case « x ». Une règle transforme un état en un autre, la **Table 1** ci-dessus représente les règles de changement d'état pour un taquin de n\*n cases :

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | | Les règles | **Pré condition** | Action | | **↑ déplacer la case x vers le haut** | x n’est pas sur la ligne 1 | Permuter la case x | | **↓ déplacer vers le bas** | x n’est pas sur la ligne n | Permuter la case x | | **→ déplacer vers la droite** | x n’est pas sur la colonne n | Permuter la case x | | **← déplacer vers la gauche.** | x n’est pas sur la colonne 1 | Permuter la case x | |

**Tableau 1 :** Les règles de changements d’états pour un taquin de n\*n cases.

L’action de chaque règle est de permuter la case x avec la case adjacente .

### **2.3 La solution :**

La solution à ce problème est **une séquence de déplacements** qui transforme l’état initial jusqu’à trouver l’état but, déduit à partir de l’espace de recherche obtenu suivant une stratégie.

## **La présentation des algorithmes de résolution et la Complexité théorique:**

Il existe plusieurs méthodes de résolution présentées dans la littérature ou on peut trouver pour un premier temps deux catégories qui nous intéressent : méthodes aveugle et méthodes heuristiques.

Dans notre étude, on va aborder la méthode de recherche en largeur d’abord et en profondeur d’abord comme méthodes aveugle et A\* comme méthode heuristique avec deux heuristiques différentes : cases mal placées et sommes dès distance absolu ou Manhattan.

### **3.1 Algorithme de résolution en largeur d’abord ou BFS:**

#### 3.1.1 Explication de fonctionnement de l'algorithme:

* Les nœuds de **OUVERT** sont les nœuds feuilles de l’arbre de recherche ou d’une autre manière le nœud qui n'ont pas encore été sélectionnés pour être développés).
* Les nœuds de **FERME** sont soit des feuilles qui n'ont pas produits de successeurs soit les nœuds qui ne sont pas des feuilles).

Nous avons en entrée l'état initial **S**, un ensemble des état finaux qui en la note **F**, Cette algorithme a comme sortie soit une solution si elle existe sinon un message d'échec. L’ensemble des nœuds ouvert ainsi l’ensemble de nœuds fermés sont initialisés à vide. Ces derniers sont représentés à l’aide de deux files chacun.

#### 3.1.2 Le pseudocode de l'algorithme:

|  |
| --- |
| **Algorithme** BFS ;  **entrée :**  s : état initial ; F : ensemble des états finaux ;  Ouverte, Fermée : file de nœuds initialement vide ;  **sortie** : une solution si succès sinon échec ;  **Début**      Insérer le nœud initial s dans la file Ouverte ;      L : **Si** Ouverte est vide **alors** échec **sinon** continuer ;      Défiler n le premier nœud de la file Ouverte et l'insérer dans Fermée ;  **Si** il n 'existe pas de successeur **alors** aller à L ;      Déterminer les successeurs de n et les enfiler dans Ouverte ;      Créer un chaînage de ces nœuds vers n ;  **Si** parmi les successeurs, il existe un état final **alors**      Succès : la solution est la chaîne des   nœuds allant du nœud courant à la racine,  **Sinon** aller à L ;  **Fin.** |

**Figure 4 :** Algorithme de résolution en largeur d’abord.

#### 3.1.3 Exemple de résolution:

Les nœuds de Ouvert sont triés dans l’ordre croissant de leur profondeur. Les moins profonds sont placés en premier. Il est montré que cette recherche garantit la découverte du chemin le plus court s’il existe. Dans le cas où il n’existe pas elle échouera pour les graphes finis et boucle pour les graphes infinis. La figure suivante (**figure 5**) représente un exemple de résolution en largeur d’abord.

|  |
| --- |
|  |

**Figure 5**: Exemple de résolution de l’algorithme de recherche en largeur d’abord.

|  |
| --- |
|  |

**Figure 6:** Exemple de résolution de l’algorithme de recherche en largeur d’abord cas de problème de taquin.

#### 3.1.4 Etude de Complexité :

La complexité est mesurée par :

-b : branchement maximal de l’arbre de recherche (nb max de successeurs à un nœud)

-d : profondeur de la meilleure solution.

**Complexité temporelle :**

1+b+b²+b3+…+bd =**O(bd)**

Supposons un espace d’état où chaque état a **b** successeurs

La racine a **b** successeur, chaque nœud au niveau suivant a de niveau **b** successeur

Au pire cas, développer tout jusqu’au dernier nœud à la profondeur **d**

**Complexité spatiale :**

**O(bd)** car on garde tous les nœuds en mémoire.

### **3.2 Algorithme de résolution en profondeur d’abord ou DFS:**

#### 3.2.1 Explication de fonctionnement de l'algorithme:

Nous avons en entrée l'état initial **S**, un ensemble des état finaux qui on le note **F**, Cette algorithme a comme sortie soit une solution si elle existe sinon un message d'échec. L’ensemble des nœuds ouvert ainsi que l’ensemble de nœuds fermés sont initialisés à vide. Ces derniers sont représentés à l’aide de deux files chacun.

Les nœuds de Ouvert sont ordonnés en ordre décroissant suivant leur profondeur dans l'arbre. Les plus profonds sont placés en premier. Ceux de même profondeur sont classés arbitrairement. En général dans cette stratégie, on impose une **limite de profondeur** pour empêcher que le système boucle sur des chemins infinis.

Au contraire de méthode de retour en arrière chronologique, Les chemin contenant les nœuds fermés (déjà développé) sont gardés en mémoire qui est l'inconvénient de cette méthode.

#### 3.2.2 Le pseudocode de l'algorithme:

|  |
| --- |
| **Algorithme DFS;  entrée:** s: état initial; F: ensemble des états finaux ;  Ouverte**:** pile de nœuds initialement vide ;  Fermée : file de nœuds initialement vide;  **sortie :** une solution si succès sinon échec ;  **Début**  empiler le nœud initial s dans Ouverte;   L: **si** Ouverte est vide **alors** échec sinon continuer;  Dépiler n, le premier nœud de Ouverte et l'enfiler dans Fermée; **si** la profondeur de l'arbre est égale au seuil de profondeur  **alors aller à** L **sinon** continuer;  déterminer les successeurs de n et les empiler dans Ouverte;      créer un chaînage de ces nœuds vers n; **si** parmi les successeurs, il existe un état final **alors** succès;  la solution est obtenue en suivant le chaînage arrière de ce  nœud vers la racine,   **sinon aller à** L **fin** |

**Figure 7:**  Algorithme de résolution en profondeur d’abord.

#### 3.2.3 Exemple de résolution:

Un exemple de l'algorithme de résolution en profondeur est présent dans la **figure 08**. Avec l’ordre A, B, D, H, I, E, J, K, C, F, L, M, G, N, O.

|  |
| --- |
|  |

**Figure 8:**  Exemple de résolution de l’algorithme de recherche en profondeur d’abord.

#### 3.2.4 Complexité :

**Complexité en temps :** b\*b...\*b m fois donc **O(bm)**

**Complexité en espace :** b+b+.... +b donc **O(b\*m)**

Optimal : Non

### **3.3 Algorithme de résolution avec A\***

#### 3.3.1 Explication de fonctionnement de l'algorithme:

La recherche heuristique représente une technique alternative de résolution de problèmes en intelligence artificielle, largement utilisée pour les problèmes qui sont caractérisés par une explosion combinatoire d'états. Les techniques de recherche heuristiques utilisent un ensemble d'information qui permet d'évaluer la probabilité qu'un chemin allant du nœud courant au nœud cible soit meilleur que les autres.

L'algorithme A\* utilise la fonction heuristique suivant f(n) = g(n) + h(n) Où n est un nœud représentant un état du problème, g(n) est le coût de la chaîne allant de l'état initial s à n et h(n) appelée heuristique est une estimation du coût de la chaîne reliant n à un état final. Cette heuristique estime le coût total de la chaîne entre l'état initial et un état cible qui passe par n.

**Heuristique 1 : Cases mal placées**

Pour tout nœud n développé lors du déroulement de l'algorithme A\*, g(n) est la longueur de la chaîne entre la racine et n et h(n) compte le nombre des chiffres mal placés pour le nœud n.

**Heuristique 2 : sommes des distances absolues (Manhattan)**

h(n) est la somme des distances entre chaque case et sa position de but.

#### 3.3.2 Le pseudocode de l'algorithme A\*:

|  |
| --- |
| Algorithme A\*;  **entrée:** s: état initial; F: ensemble des états finaux ; c(n, n) le coût entre les nœuds adjacents n, et n; h: fonction heuristique; Ouverte: liste de nœuds triée selon la fonction f et initialement vide; Fermée : file de nœuds initialement vide; **sortie :** une solution si succès sinon échec  **début**  f(s):= g(s) +h(s);  insérer le nœud initial s dans la liste Ouverte avec f(s);  **tant que** Ouverte n'est pas vide **faire**  **début**      retirer n le premier nœud de la liste Ouverte     qui a la plus petite valeur f     en cas de conflit, choisir arbitrairement n;     enfiler n dans Fermée ;     si n est un état final alors succès: la solution est obtenue     en suivant le chaînage arrière de n vers la racine     **sinon si** na des successeurs **alors**    **pour** **chaque successeur n faire**        **début** f(n):= g(n) + h(n);        si n, n'est ni dans Ouverte ni dans Fermée alors              **début**              enfiler n, avec fin) dans Ouverte selon l'ordre             croissant de f             créer un chaînage de n, vers n:             **fin**       **sinon si** f(n) est inférieure à la valeur de n        dans Ouverte ou Fermée **alors** début remplacer cette valeur par fin):       mettre à jour le chaînage arrière :       **fin**     **fin**   **fin** **fin** |

**Figure 9:** Algorithme de résolution avec A\*

#### 3.3.3 Exemple de résolution (case mal placé):

|  |
| --- |
| **2 8 3 1 2 3**  **1 6 4 8 0 4**  **7 0 5 7 6 5**  **ETAT INIT ETAT BUT**  Le Coût est représenté en **noir**, et le nombre de cases mal placées en **rouge** |

**Figure 10 :** Exemple de résolution de l’algorithme de recherche A\* h1 (case mal placé)

#### 3.3.4 Exemple de résolution (Manhattan):

|  |
| --- |
| **2 8 3 1 2 3**  **1 6 4 8 0 4**  **7 0 5 7 6 5**  **ETAT INIT ETAT BUT**  Le Coût est représenté en **noir**, et la somme des distances de Manhattan en **rouge**  Aucune description disponible. |

**Figure 11 :** Exemple de résolution de l’algorithme de recherche A\* h2 (Manhattan)

#### 3.3.5 Complexité :

Temporel: O(bd) polynomial si h est optimal

Spatial: O(bd)

La figure suivante présente un résumé des complexités des différents algorithmes:

|  |
| --- |
|  |

**Figure 12 :** Comparaison entre les différents algorithmes

## **Modélisation et structures de données :**

### **Représentation d’un état :**

L’espace de recherche est modélisé sous forme d’arbre binaire.

Un nœud de l’arbre représente un état, un état peut être une solution (complète ou partielle), l’état est principalement constitué de :

- Une chaîne de caractères déjà instanciées ou pas, représentant la solution (complète ou partielle)

- Son nœud père

- Une liste contenant ses fils

- La direction que peut prendre cet état

- Une variable indiquant son niveau dans l’arbre

- Le coût estimé pour arriver au but

Cette structure d’état est commune aux algorithmes de parcours en largeur d’abord et en profondeur d’abord, l’attribut Coût a été rajouté pour l’algorithme A\*, cet attribut dépend des heuristiques utilisées par ce dernier.

### **Génération de l’état initial:**

On génère l’état initial aléatoirement, en fixant un nombre donné de carreaux à leurs places selon le niveau de difficulté souhaité (le niveau facile par exemple aura 5 nombres bien placés, le moyen 3 nombres et le difficile 0).

Après la génération, un test de **solvabilité** est effectué.

#### **Test de solvabilité :**

Ce test est nécessaire car les états initiaux où le nombre d'inversions est impair rendent le jeu impossible.

Le nombre d’inversions c’est le nombre total de couples (a, b) où **a** est placé avant **b** dans la liste ET **a** est supérieur à **b**.

Par exemple en lisant notre état but, ligne à ligne, de haut en bas les nombres devrait être placé comme suit :

L’emplacement du 1 devrait être inférieur à celui des autres

L’emplacement du 2 devrait être inférieur à celui des {3, 8, 0, 4, 7, 6, 5}

L’emplacement du 4 devrait être inférieur à celui des {7, 6, 5}

et ainsi de suite..

Pour rendre un état **Insolvable -> Solvable :**

On pourra changer la parité du nombre d’inversions en échangeant un couple (a, b) présent dans la liste (comme suit : ... a ... b ...), tel que si a > b on aura une inversion de moins.

### **Exploration des états:**

Pour faire la recherche, 2 structures de données sont utilisées pour parcourir les états du problème.

Une liste **Ouverte** dont l’ordre de tri, décide de l’ordre d’exploration des nœuds, et une autre **Fermé** dans laquelle on stock les nœuds déjà visités.

La politique d’utilisation de la liste **Ouverte** diffère d’un algorithme à un autre :

- DFS gère sa liste **Ouverte** selon la politique LIFO.

- BFS gère sa liste **Ouverte** selon la politique FIFO.

- A\* gère sa liste **Ouverte** selon les priorités calculées grâce à l’évaluation heuristique.

Comme le nombre des états est prohibitif, pour rechercher un état dans ces structures, il est nécessaire d’utiliser une fonction de hachage.

Pour le faire on a déclaré notre liste **Fermé** comme <HashMap> et redéfini la fonction hashCode() en suivant la méthode qu’a utilisé *Josh Bloch* dans son livre *Effective Java* ***[2]****,* comme suit :

|  |
| --- |
| **Procédure hashCode;**  **entrée :** Etat : String; // l'état actuel du taquin sous forme d'une chaîne de caractères  **sortie :** résultat : entier ; **var :** résultat = 17 ;  **Début** résultat = 37 \* résultat + Etat; **Fin** |

**Figure 13 :** Algorithme de la fonction de hachage

*Remarque : 17 et 37 ont été choisis car ce sont des nombres premiers impairs, cette propriété minimise le risque de collision.*

### **Fonction de successeur:**

Si l’état actuel ne correspond pas à l’état but, la fonction successeur génère une liste d'états qui peuvent être atteint selon les actions permises.

### **Actions :**

Il y a 4 actions possibles correspondant aux quatre façons de changer la position du carré vide : haut, bas, gauche, droite

## **Présentation de l’Interface graphique**

Cette section est dédiée à la présentation de l’interface de notre logiciel qui a été conçu avec le langage orienté objet JAVA (JDK, version 8) en utilisant la bibliothèque SWING , dans ce qui suit on va présenter les différentes fonctionnalités et l’interaction homme/machine.

La figure suivante représente la page d'accueil de notre programme. En cliquant sur le bouton « afficher le Solveur » vous serez dirigé vers la page principale.

|  |
| --- |
|  |

**Figure 14 :** Page d’accueil

La figure suivante (**Figure 15**) représente la page principale de notre programme. L'expérience utilisateur a une grande importance dans notre travail. Pour l’utiliser, on procède comme suit :

* Choisir une méthode de résolution à gauche (DFS, BFS, A\* avec deux options h1 et h2) (**Figure 16**)
* Choisir une difficulté (**Figure 17**)
* Cliquez sur « Résoudre » pour démarrer la recherche (voir les figures ci-dessous).

|  |
| --- |
|  |

**Figure 15 :** Page principale

|  |
| --- |
|  |

**Figure 16 :** Méthodes de résolution

|  |
| --- |
|  |

**Figure 17 :** Niveaux de difficulté

|  |
| --- |
|  |

**Figure 18 :** Page montrant les résultats après résolution

Une fois la résolution finie une 3ème fenêtre apparaît, nous permettant ainsi de visualiser les différents mouvements qu’a suivi le solveur jusqu’à obtention du but. Deux boutons sont disposés pour faciliter la navigation entre l'état suivant et le précédent. Voir figure suivante:

|  |
| --- |
| Etat initial ……… Etat pris du milieu ……… Etat Final |

**Figure 19 :** Visualisation du déroulement de la recherche

**Démonstration :**

Une petite vidéo qui présente les différentes fonctionnalités de notre solveur :

[**https://www.youtube.com/watch?v=0uZmRmLhptw**](https://www.youtube.com/watch?v=0uZmRmLhptw)

## **Etude Expérimentale:**

**Environnement du travail;**

*Matériel :*

*L'implémentation de l’algorithme a été réalisée sur un micro-ordinateur ayant la configuration*

*matérielle suivante :*

*- Un micro processor Intel® Core™ i5 1035g1 1.0 ghz boost 3.6 ghz.*

*- Une mémoire de 8.00 GB de RAM.*

*- Windows 10 édition intégrale 64 bit.*

*Logiciel :*

*Afin d'implémenter notre algorithme, nous avons opté pour :*

*Le langage Java comme langage de programmation.*

*Java swing pour le développement de l’interface graphique*

### **6.1. Temps d'exécution (en millisecondes):**

Le tableau suivant présente les résultats de temps d’exécution qu’on a obtenus en variant la configuration initiale du taquin.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Configuration** | **BFS** | **DFS** | **Mal placé** | **Manhattan** |
| 218347650 | 603 | 157073 | 46 | 34 |
| 723804615 | 448 | 2020 | 295 | 203 |
| 143208765 | 1115 | 7895 | 71 | 55 |
| 143208765 | 1163 | 9140 | 109 | 88 |
| 153204768 | 2599 | 8120 | 6388 | 3883 |
| 180234765 | 2543 | 177947 | 239 | 189 |
| 832104765 | 2258 | 71018 | 270 | 173 |
| 153824076 | 282 | 133664 | 94 | 64 |
| 125403867 | 12650 | 12286 | 2450 | 1341 |
| 128704365 | 10056 | 170718 | 11147 | 3632 |
| **La moyenne** | **3371,7** | **74988,1** | **2110,9** | **966,2** |

**Tableau 2**: Le temps d'exécution obtenus par les quatre méthodes de résolution

### **Nombre de nœuds développés:**

Le tableau suivant contient nombre de nœuds développés par les différentes méthodes, obtenus en variant la configuration initiale du taquin.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Configuration** | **BFS** | **DFS** | **Mal placé** | **Manhattan** |
| 218347650 | 3325 | 64316 | 437 | 290 |
| 723804615 | 2175 | 5874 | 2140 | 1312 |
| 143208765 | 5598 | 13236 | 574 | 367 |
| 143208765 | 5598 | 13236 | 547 | 367 |
| 153204768 | 9878 | 13253 | 21618 | 12188 |
| 180234765 | 10771 | 142931 | 1932 | 1261 |
| 832104765 | 10183 | 42040 | 1725 | 1150 |
| 153824076 | 1515 | 175095 | 386 | 246 |
| 125403867 | 30516 | 30516 | 11038 | 5874 |
| 128704365 | 28098 | 146528 | 27897 | 13770 |
| **La moyenne** | **10765,7** | **64702,5** | **6829,4** | **3682,5** |

**Tableau 3 :** Le nombre de nœuds développés par les quatre méthodes de résolution

### **6.3. La profondeur :**

Le tableau suivant présente la profondeur atteinte par chaque méthode, en variant la configuration initiale du taquin.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Configuration** | **BFS** | **DFS** | **Mal placé** | **Manhattan** |
| 218347650 | 15 | 55125 | 15 | 15 |
| 723804615 | 13 | 5723 | 13 | 13 |
| 143208765 | 15 | 12875 | 15 | 15 |
| 143208765 | 15 | 12875 | 15 | 15 |
| 153204768 | 17 | 12889 | 17 | 17 |
| 180234765 | 17 | 43155 | 17 | 17 |
| 832104765 | 17 | 39023 | 19 | 17 |
| 153824076 | 13 | 7817 | 13 | 13 |
| 125403867 | 19 | 19 | 19 | 19 |
| 128704365 | 19 | 40157 | 21 | 21 |
| **La moyenne** | **16** | **22965,8** | **16,4** | **16,2** |

**Tableau 4** : La profondeur atteinte par les quatre méthodes de résolution

### **6.3. Graphes**:

|  |
| --- |
|  |

**Figure 20 :** graphique comparatif du temps d'exécution obtenu par les quatre méthodes de résolution

Analyse :

DFS prend beaucoup de temps lors de son exécution, tandis que BFS et les deux heuristiques de A\* s'exécute plus rapidement.

On remarque aussi que l'heuristique Manhattan est légèrement meilleure en la testant avec une configuration difficile.

|  |
| --- |
|  |

**Figure 21 :** graphique comparatif de la profondeur atteinte par les quatre méthodes de résolution

Analyse :

On remarque que les seuils de BFS et des deux heuristiques (mal placé et Manhattan ) reste négligeable devant les seuils atteints par DFS quel que soit la configuration initiale.

|  |
| --- |
|  |

**Figure 22 :** graphique comparatif du nombre de nœuds développés par les quatre méthodes de résolution

Analyse :

DFS génère un nombre élevé de nœuds, tandis que BFS et les deux heuristiques de A\* développent moins de nœuds.

Les résultats suggèrent que l’heuristique Manhattan est meilleure que celle des cases mal placées, on peut dire que Manhattan la domine car elle développe moins de nœuds.

### **Comparaison entre méthode :**

D'après les graphes comparatifs du **temps d'exécution** obtenu et du **nombre de nœuds** développés par les quatre méthodes de résolution, on remarque que les deux heuristiques (mal placé et Manhattan ) performent mieux que BFS et DFS donc on peut dire que les méthodes heuristiques sont plus rapide que les méthodes aveugles, et l’heuristique Manhattan est la meilleure.

Par conséquent, on conclut que l’algorithme A\* est le mieux adapté à ce problème. La méthode BFS peut être rapide dans les cas simples, mais dans les cas les plus complexes, A\* est le plus efficace (avec l’heuristique Manhattan), c’est donc lui le meilleur candidat pour trouver une solution en temps raisonnable.

## **Conclusion**

Dans cette première partie de TP nous avons pu faire une recherche sur le problème de taquin et appliquer les différentes résolutions vues en cours en utilisant un langage de programmation (java), faire une étude de comparaison entre les différentes méthodes et enfin créer une interface graphique pour présenter notre travail.

Il est évident d'après les résultats empiriques que les méthodes heuristiques (mal placés et Manhattan) sont mieux que les méthodes aveugles (DFS BFS) en matière de temps d'exécution et nombre de nœuds à développer. Selon les résultats obtenus, on conclut que les méthodes heuristiques sont plus efficaces que les méthodes dites aveugles. Nous avons démontré dans cette première partie de TP la nécessité d'utiliser des approches intelligentes pour aborder des problèmes complexes à grande échelle tels que le problème du taquin.

**Références:**

**[1]** Drias, H. Z. Algorithmique Moderne Analyse et Complexité. Alger: Office des Publications Universitaires, 2017.

**[2]** Effective Java™: Programming Language Guide. by Joshua Bloch. June 2001.