# Université des Sciences et de la Technologie HOUARI BOUMEDIENE Département d'informatique



TP

META-HEURISTIQUE

# « PROJET DE REALISATION

# DU TAQUIN (3X3) »

***OBJECTIFS***

Dans ce Projet, on va réaliser un programme pour résoudre le jeu de taquin de (3X3).

* La résolution du problème se fait par 3 approches différentes :
  1. L’algorithme BFS
  2. L’algorithme DFS
  3. L’algorithme A\* (avec nombre case mal places comme heuristique)
  4. L’algorithme A\* (avec distance vers état but comme heuristique)
* Les outils utilisés pour réaliser ce projet sont :
  1. Langage de programmation Java
  2. L’interface graphique JavaFX

***Introduction***

En intelligence artificielle, la résolution de problèmes par la recherche comprend la formulation d'un problème comme une question de recherche, puis construire sur cette formule un arbre de recherche qui sera parcouru pour trouver la cible. Ce processus de parcours utilise des techniques de recherche qui peuvent être divisées en deux grandes catégories. Les techniques de recherche non informées tels que le Breath First Search ou le Depth First Search et techniques de recherche informées conçu à l’aide d’heuristiques tels que l’algorithme de recherche A\*.

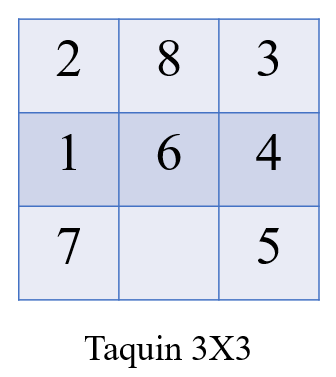
Les techniques de recherche non informées incluent l'exploration aveugle d'un arbre de recherche pour trouver l'état cible. En fait, il s'agit d'explorer les voies possibles à partir

Un point de départ (état initial) pour déterminer si une des voies explorées conduit à un état cible. Ces techniques sont parfois très coûteuses et même impossibles, selon le nombre d'états dans l'arbre de recherche. Voici pourquoi des techniques de recherche informée ont été développées.

# Introduction sur problème du Taquin (3x3) :

Le taquin est un puzzle de tuiles coulissantes composé d'une zone de cadre à structure carrée contenant des tuiles dans un ordre aléatoire/irrégulier avec une tuile manquante. C'est une version plus petite du 15-puzzle.Le taquin est essentiellement une zone de cadre séparée en grilles 3x3 contenant 8 tuiles et une grille vide. Les carreaux sont marqués d'une certaine manière afin qu'ils puissent être identifiés. Les tuiles sont majoritairement numérotées de 1 à 8. On nous donne une configuration initiale des tuiles. Une configuration finale souhaitée est également donnée. Nous devons atteindre l'état final en faisant glisser les tuiles à l'aide de la grille vide présente.

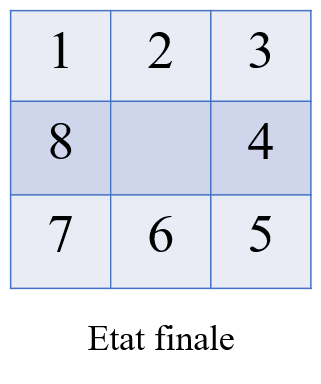
dans la figure ci-dessous un exemple d’un taquin (3X3) :



La **configuration initial** du taquin peut être **quelconque**, et le but est de déplacer les cases jusqu’à obtenir généralement une **configuration ordonné** (1 – 2 – 3 – 4 – 5 – 6 – 7 – 8 – 0)

Avec le **0** désigne **la case vide**.

Exemple :



# Définition des espaces d’états pour problème du Taquin (3x3) :

Le nombre de configurations du jeu du taquin 3x3 est égal au nombre de toutes les combinaisons (3x3) possibles , donc :

Nb\_comb\_taquin(3X3) = 9! = **362880 combinaisons**

**Les états du problème**:  
tous les états possible : 362880 états.

**L’état initial :**

C’est un état quelconque,

**L’état final :**

Généralement c’est l’état ordonne, c.à.d.

(1 – 2 – 3 – 4 – 5 – 6 – 7 – 8 – 0) .

Mais ça peut être n’importe quel état diffèrent de l’état initial.

**Les opérateurs :**

Permet d’aller d’un état a un autre état, les opérateurs possibles sont :

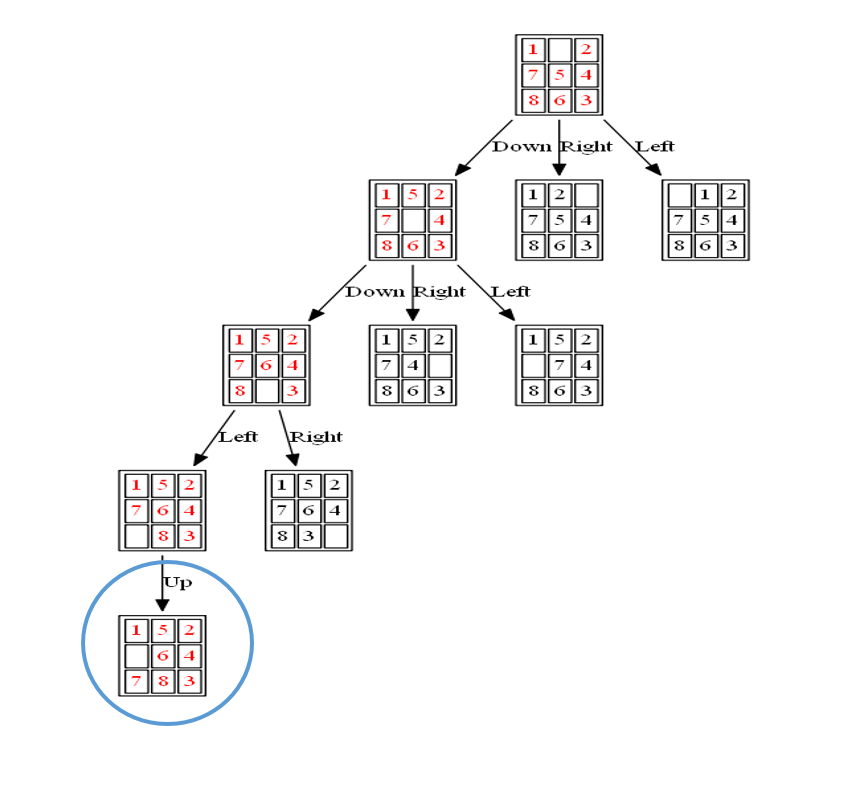
1. Déplacer case vide en **HAUT**
2. Déplacer case vide à **DROITE**
3. Déplacer case vide en **BAS**
4. Déplacer case vide à **GAUCHE**

Exemple :

# 

# Propriétés sur problème du Taquin (3x3) :

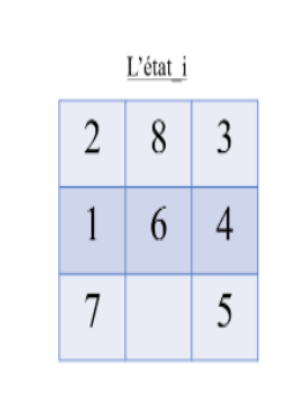
Pendant la résolution du jeu du taquin, il ne faut pas explorer une combinaison **plus qu’une fois** car ça nous mènera à **une boucle infinie** pendant la recherche de la solution, comme représenter dans la figure ci-dessous :



La **vérification** si un état a été explorer parmi **362880** états possibles est **couteuse**, pour trouver une solution à ce problème on utilise une méthode de **hachage** pour vérifier si un état a été explorer ou pas, la méthode de hachage utiliser dans ce projet est :



Pour que la comparaison entre un état et un état but ne soit **pas couteuse** la **représentation** des états se fait sous forme **d’un entier**, par exemple : la matrice ci-dessous est représenter par un nombre **283 164 705 .**



# Résolution avec L’algorithme BFS :

## Définition du l’algorithme BFS :

La recherche **en largeur** (en anglais **Breadth First Search** **BFS**) est une technique de recherche importante dans les **graphes** (précisément les **arbres**) qui est utilisé pour résoudre de nombreux problèmes, notamment pour trouver **le chemin le plus court** dans un graphe et résoudre des **jeux de réflexion** (tels que les **Rubik's Cubes**). De nombreux problèmes en informatique peuvent être envisagés en termes de graphes. Par exemple, **l'analyse des réseaux**, **la cartographie** des routes et **l'ordonnancement** sont des problèmes de graphes. Les algorithmes de recherche sur les graphes, comme la recherche en premier, sont utiles pour analyser et résoudre les problèmes liés aux graphes.

## Principe de l’algorithme :

Une fois que l'algorithme **a visité** et **marqué** le nœud de départ **(la racine),** il **se déplace** vers les nœuds **non visités les plus proches** qui sont les **fils** et les analyse par vérifier l’obtention de **l’état but** ou dans l’autre cas les insérer dans **open**. Une fois visités, tous les nœuds sont marqués en les rajoutant dans **fermé**. Ces itérations se poursuivent jusqu'à ce qu’on atteint l**e but** ou que tous les nœuds du graphe **aient été visités** et marqués avec succès.

## Pseudo Code du l’algorithme BFS utilisé :

### Expand :

La fonction **expand()** génère tous les fils d’un nœud donné, en appelant les méthodes up, down, left et right .

**Pseudo code :**

**Entrée:** n : Node *//contient l’état, le père, move : étape allant de de père à n (up/left/down/right);*

**Sortie :** quatre nœud développer *//left, right, up et down ;*

**Var :** n.left, n.right, n.up, n.down : Node *// les quatre nœuds à développer*

**Début**

n.left = left(n) ;

*//la fonction left permet de change la position de 0 dans le nœud avec son voisin gauche (la même chose pour right (), up () et down()).*

n.left.setPere(n, ‘left’) ;

n.right = right(n) ;

n.right.setPere(n, ‘right’) ;

n.up = up(n) ;

*//la fonction setPere affecte les valeurs des varibles insérées comme paramètres*

*a l’attribut père et move dans les nœuds fils générés*

n.up.setPere(n, ‘up’) ;

n.down = down(n) ;

n.down.setPere(n, ‘down’) ;

**Fin.**

### insererFilsOuvert :

Cette méthode parcourt la file open , chaque nœud obtenu en défilant cette file , on l’insère dans la table hashtable (fermé) si il n’existe pas , par la suite elle génère ces fils et les rajoute dans open. Ce traitement se répète jusqu’à arriver au nœud but ou bien avoir la file open vide.

**Entrée :** S : String *//contient l’état final;*

n : Node *//contient l’état, le père, move : étape allant de de père à n (up/left/down/right);*

**Sortie :** nœud *But //si on n’a pas atteint l’état final on aura un null en sortie;*

**Var :** neoud : Node

continu : booléen *//permet de poursuivre*

h, nb : entier

**Début**

continu<- vrai ;

**si** (isEmpty(open)) *// il y a aucun noeud a traité donc pas de solution*

retourner null ;

**sinon**

**tant que** ( continu et !isEmpty(open))

nœud<-defiler(open) ;

nb<- la valeur du nœud en entier ;

h<- hash(nb) ;

*//augment le nombre de collision dans la ligne d’indice*

ferme[h][0]++ ;

*//ajouter la valeur de nœud dans ferme avec le hashage*

ferme[h][ ferme[h][0]]<- nb ;

expand(nœud) ;

**si** (noeud.up != null)

**si** (val(noeud.up )=goal)

nb<- la valeur du nœud.up en entier ;

h<- hash(nb) ;

**si** (ferme[h][0]=0) //*ya pas de collision dans la ligne*

*d’indice donc se nœud n’est pas visité*

ajouter dans open*;*

**sinon**

//*vérifier si ce n’apparait pas dans les collision*

*//si oui*

ajouter dans open*;*

**fin si**

**sinon** *// goal est trouvé*

continu <- faux ; *// on sort de la boucle*

retourner noeud.up ;

**fin si**

**fin si**

**si** (noeud.down != null)

**si** (val(noeud.down )=goal)

nb<- la valeur du nœud.down en entier ;

h<- hash(nb) ;

**si** (ferme[h][0]=0) //*ya pas de collision dans la ligne*

*d’indice donc se nœud n’est pas visité*

ajouter dans open*;*

**sinon**

//*vérifier si ce n’apparait pas dans les collision*

*//si oui*

ajouter dans open*;*

**fin si**

**sinon** *// goal est trouvé*

continu <- faux ; *// on sort de la boucle*

retourner noeud.down ;

**fin si**

**fin si**

**si** (noeud.right != null)

**si** (val(noeud. right )=goal)

nb<- la valeur du nœud. right  en entier ;

h<- hash(nb) ;

**si** (ferme[h][0]=0) //*ya pas de collision dans la ligne*

*d’indice donc se nœud n’est pas visité*

ajouter dans open*;*

**sinon**

//*vérifier si ce n’apparait pas dans les collision*

*//si oui*

ajouter dans open*;*

**fin si**

**sinon** *// goal est trouvé*

continu <- faux ; *// on sort de la boucle*

retourner noeud. right ;

**fin si**

**fin si**

**si** (noeud.left != null)

**si** (val(noeud. left )=goal)

nb<- la valeur du nœud. left  en entier ;

h<- hash(nb) ;

**si** (ferme[h][0]=0) //*ya pas de collision dans la ligne*

*d’indice donc se nœud n’est pas visité*

ajouter dans open*;*

**sinon**

//*vérifier si ce n’apparait pas dans les collision*

*//si oui*

ajouter dans open*;*

**fin si**

**sinon** *// goal est trouvé*

continu <- faux ; *// on sort de la boucle*

retourner noud. left ;

**fin si**

**fin si**

**fin tant que**

**fin si**

**Fin.**

### Chainage :

Une fois la solution trouvé , on obtient la chaîne des nœuds allant du nœud courant à la racine en faisant un chainage .

**Pseudo code :**

**Entrée:** n : Node *//contient l’état, le père, move : étape allant de de père à n (up/left/down/right);*

**Sortie :** pile *//contient le chemin à partir du nœud goal jusqu’à la racine ;*

**Var :** Stack : pile

**Début**

**Tant que** (n.pere != null)

empiler (stack, n.move) ; *//empiler l’étape dans stack*

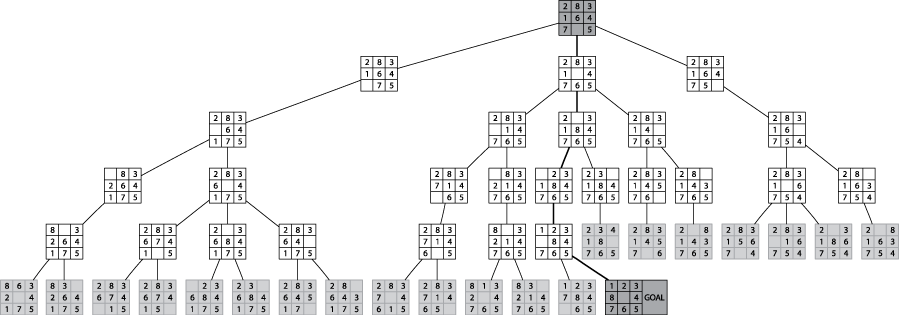
empiler (stack, n.pere) ; *//empiler le père dans stack*

n<- n.pere 

**Fin tant que**

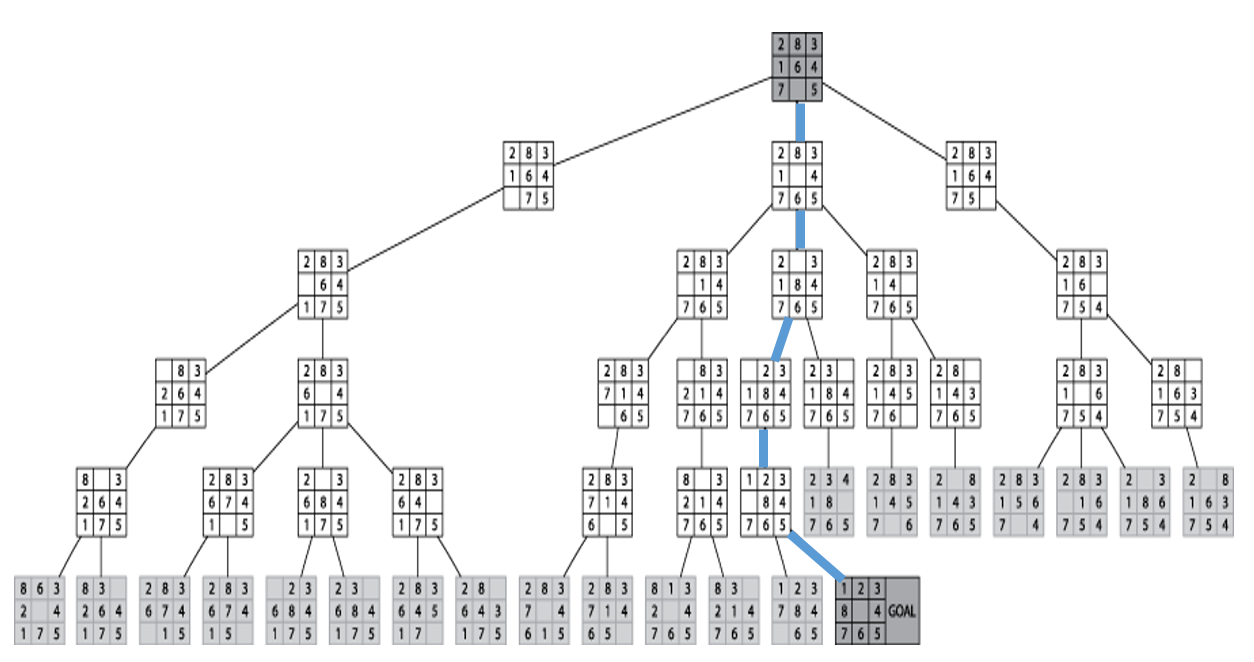
**Fin.**

Exemple :



La solution a été obtenue après explorer 5 niveaux des fils (**profondeur = 5**)

Le chemin vers la solution selon l’algorithme BFS sera :



*La solution BFS d’un Taquin (3X3) selon l’algorithme BFS*

Donc le chemin de l’état initial vers l’état final sera :

Déplacer la case vide en **HAUT**

Déplacer la case vide en **HAUT**

Déplacer la case vide vers la **GAUCHE**

Déplacer la case vide en **BAS**

Déplacer la case vide vers la **DROITE**

Donc le chemin est : **UP – UP – LEFT – DOWN - RIGHT**

**Compléxité temporelle et spatiale :**

Le Breadth first search est complet si b (le facteur de branchement maximal) est fini, ce qui est souvent le cas dans la plupart des problèmes courant dont le n-puzzle. La complexité en temps de cet algorithme est O(bd) . En effet, l’algorithme crée et examine tous les noeuds de profondeur au plus d et en plus ,lors de l’examen des noeuds de profondeur d, crée tous les noeuds de profondeur d + 1 sauf B noeuds qui représentent les successeurs du noeud but. La complexité en espace est aussi de O(bd) . Tous les noeuds générés sont gardés en mémoire au cours de l’exécution du Breadth first search. La mémoire est donc un véritable problème. Il est optimal si le coût de chaque action est de 1 .

Test de l’exemple précédant dans l’Application :

# 2022-04-08_163947

Déplacer la case vide en HAUT on obtient : 283104765

Déplacer la case vide en HAUT on obtient : 203184765

Déplacer la case vide à GAUCHE on obtient : 023184765

Déplacer la case vide en BAS on obtient : 123084765

Déplacer la case vide à DROITE on obtient : 123804765

Le but est **atteint !**

# Résolution avec L’algorithme DFS :

## Définition du l’algorithme DFS :

L'algorithme de parcours **en profondeur**ou**DFS** (**Depth-First Search**) est un [algorithme](https://fr.wikipedia.org/wiki/Algorithme) de [parcours **d'arbre**](https://fr.wikipedia.org/wiki/Parcours_d%27arbre), et plus généralement de [parcours **de graphe**](https://fr.wikipedia.org/wiki/Parcours_de_graphe). Il se décrit naturellement de **manière**[**récursive**](https://fr.wikipedia.org/wiki/Algorithme_r%C3%A9cursif). Son application la plus simple consiste à déterminer s'il existe **un**[**chemin**](https://fr.wikipedia.org/wiki/Chemin_(th%C3%A9orie_des_graphes))**d'un sommet à un autre**. Le nom d'algorithme en **profondeur** est dû au fait que, **contrairement** à l'[**algorithme de parcours en largeur**](https://fr.wikipedia.org/wiki/Algorithme_de_parcours_en_largeur)**,** il explore en fait **« à fond »** les chemins **un par un** : pour chaque sommet, il marque **le sommet actuel**, et il prend le **premier sommet voisin** jusqu'à ce qu'un sommet n'ait **plus de voisins** (ou que tous ses voisins soient marqués), et revient alors **au sommet père**.

**Principe de l’algorithme :**

Dans cette algorithme le choix du nœud à développer sera **le plus récent développé**, les autres nœuds seront **stockés dans une pile** de nœuds jusqu’à atteindre le **seuil de profondeur** si on a toujours **pas trouvé le but** on fait un **retour arrière vers le plus récent nœud** non encore développé et ainsi de suite jusqu’on a trouvé l’état but --> on arrête avec sortie **GOAL FOUND** et on affiche le chemin pour arriver à la matrice but sinon on continue cette procédure jusqu’à développer tous les nœuds à développer jusqu’au seuil de profondeur donné et on arrête avec **GOAL NOT FOUND**).

## Pseudo Code du l’algorithme DFS utilise :

Les fonctions qu’on a utilisées:

### Expand ():

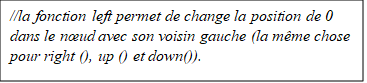
**Entrée:** S : String *//contient l’état final;*

n : Node *//contient l’état, l’heuristique, la profondeur et le cout ;*

**Sortie :** quatre nœud développer *//left, right, up et down ;*

**Var :** n.left, n.right, n.up, n.down : Node *// les quatre nœuds à développer*

**Debut**

n.left = left(n) ;

n.right = right(n) ;

n.up = up(n) ;

n.down = down(n) ;

**Fin.**

### Profondeur ()

**Entrée :** S: String *//contient l’état final;*

n : Node *//contient l’état, la profondeur du nœud courant*

Antecedent[][]: *on va stocker le id de chaque nœud développé avec celle de son parent ;*

Pile //pour stocker les fils

Prof : *limite de profondeur*

**Sortie :**Nœud à développer à chaque appelle de la fonction

**Var :** goal :boolean *//vérifier si on a attient le but*

not\_found : boolean *//permet de sortie de la boucle une la profondeur attient*

**Début**

Goal=faux;

not\_found=faux;

n.prof=0; *// initialisation de la profondeur du 1er nœud =0*

Ajouter le noeud n a la table Antecedent

Tant que (goal ==false et not\_found==false)

**Début**

Si(n.prof<prof)

**Début**

*//incrementer n.prof pour affecter la profondeur incrémenter aux fils*

expand(n);

n.prof++;

Si nœud développer (np) !=null

**Début**

si np n’existe pas dans Antecedent[][]

**Début**

Ajouter np à Antecedent[][]

np.prof=n.prof;

Empiler (np) dans la pile;

**Fin si**

**Fin si**

**Fin si**

Si (la pile n’est pas vide)

**Début**

n=sommet de la pile; //nouveau nœud à développer

**Fin**

Sinon

**Début**

Not\_found=true;//on a dépiler tous les nœuds et on les a développer jusqu’a profondeur atteinte donc on sort de la boucle sans atteindre le but

**Fin**

Si (n.noeud == S)

**Début**

//si le nœud courant == le nœud finale

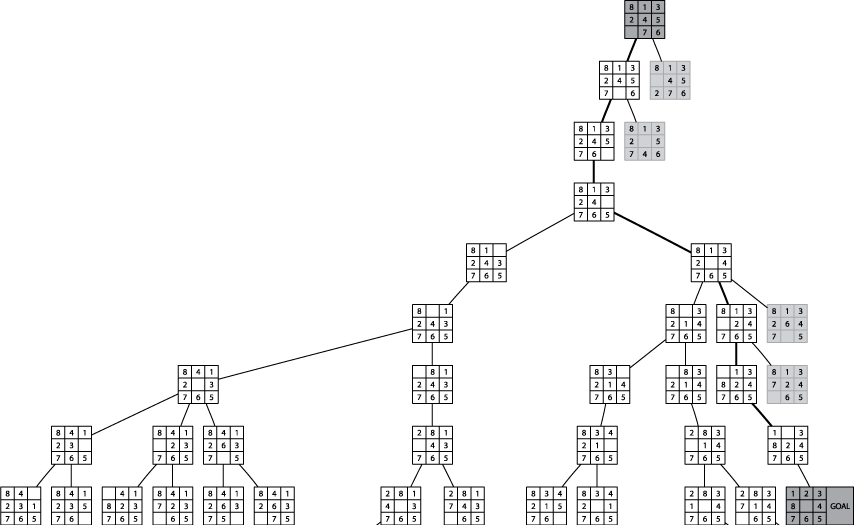
Goal==true; *//pour quitter la boucle*

**Fin**

**Fin tant que**

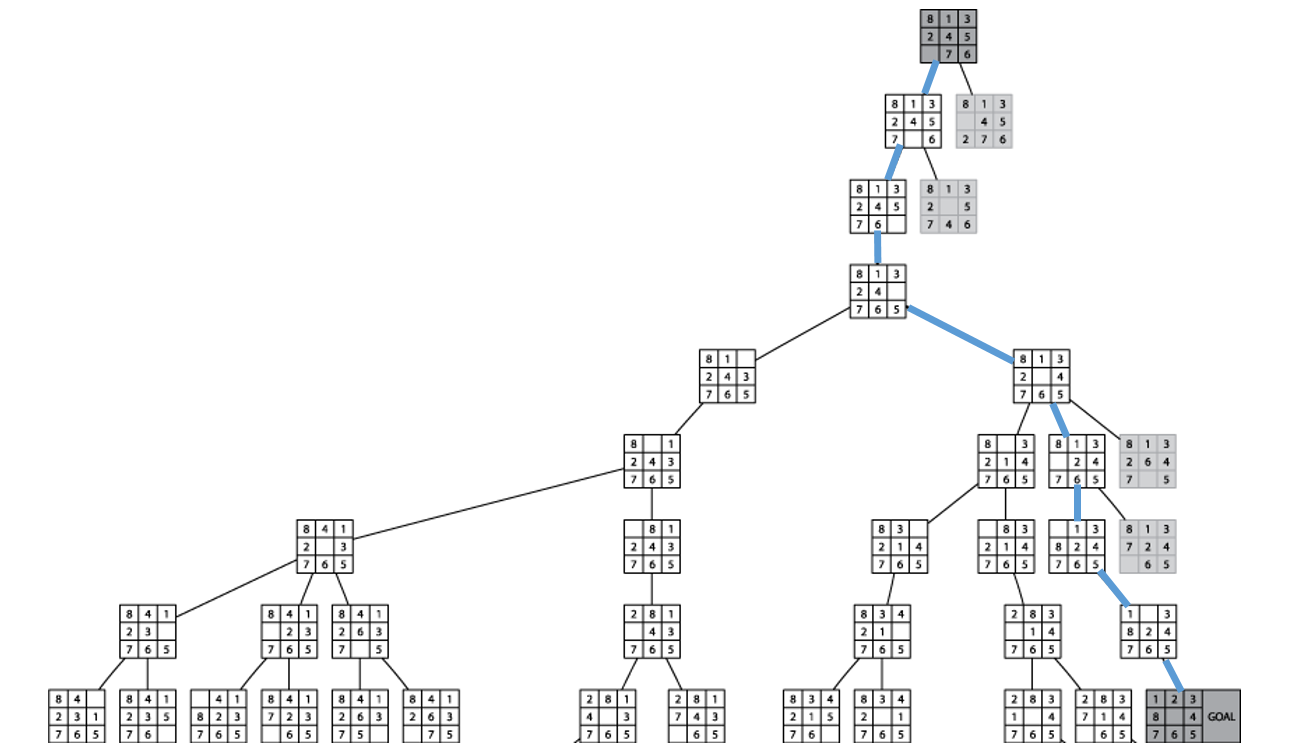
**retourner n;**

**Fin.**



**Exemple du déroulement de DFS:**

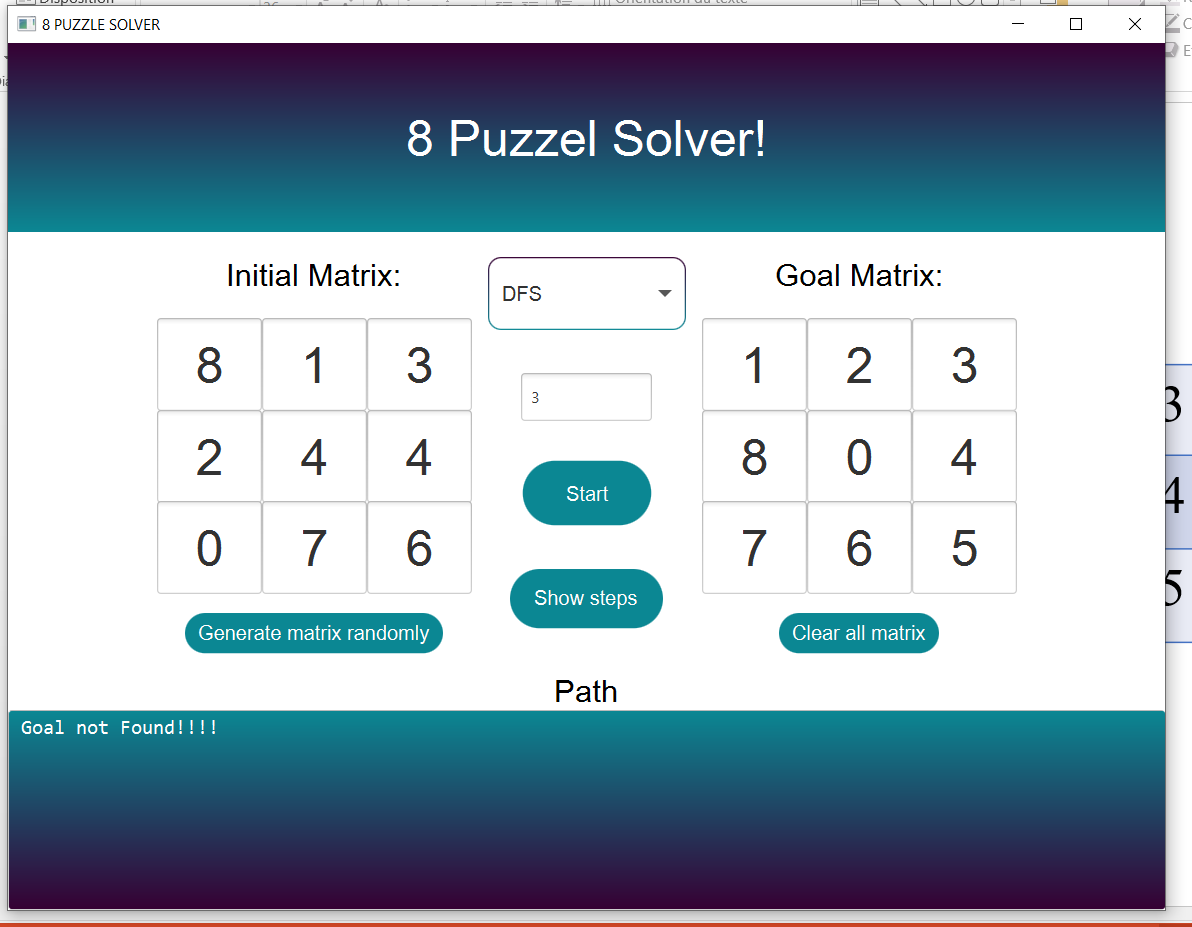
Le chemin vers la solution selon l’algorithme DFS sera :



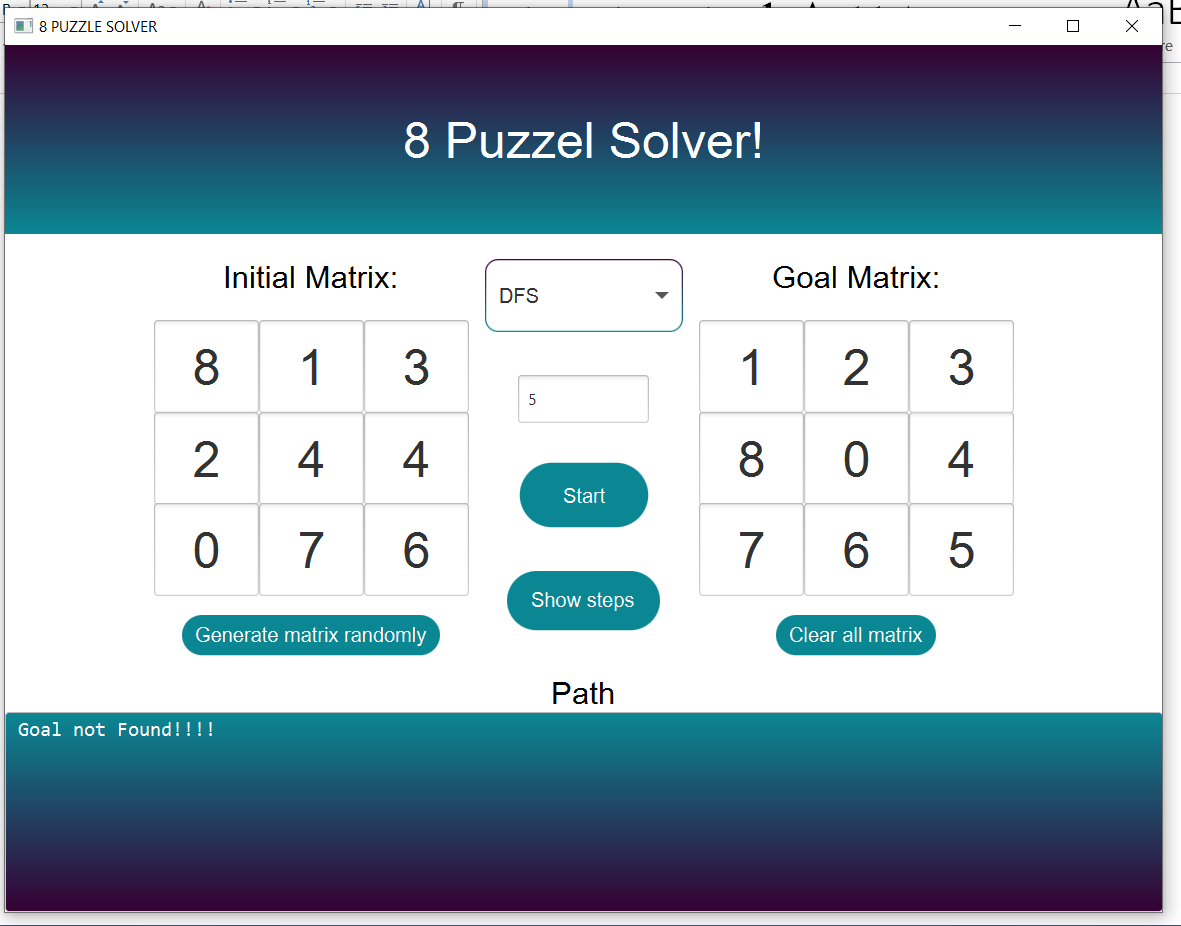
*La solution d’un Taquin (3X3) selon l’algorithme DFS*

Test du l’exemple précédant dans l’Application :

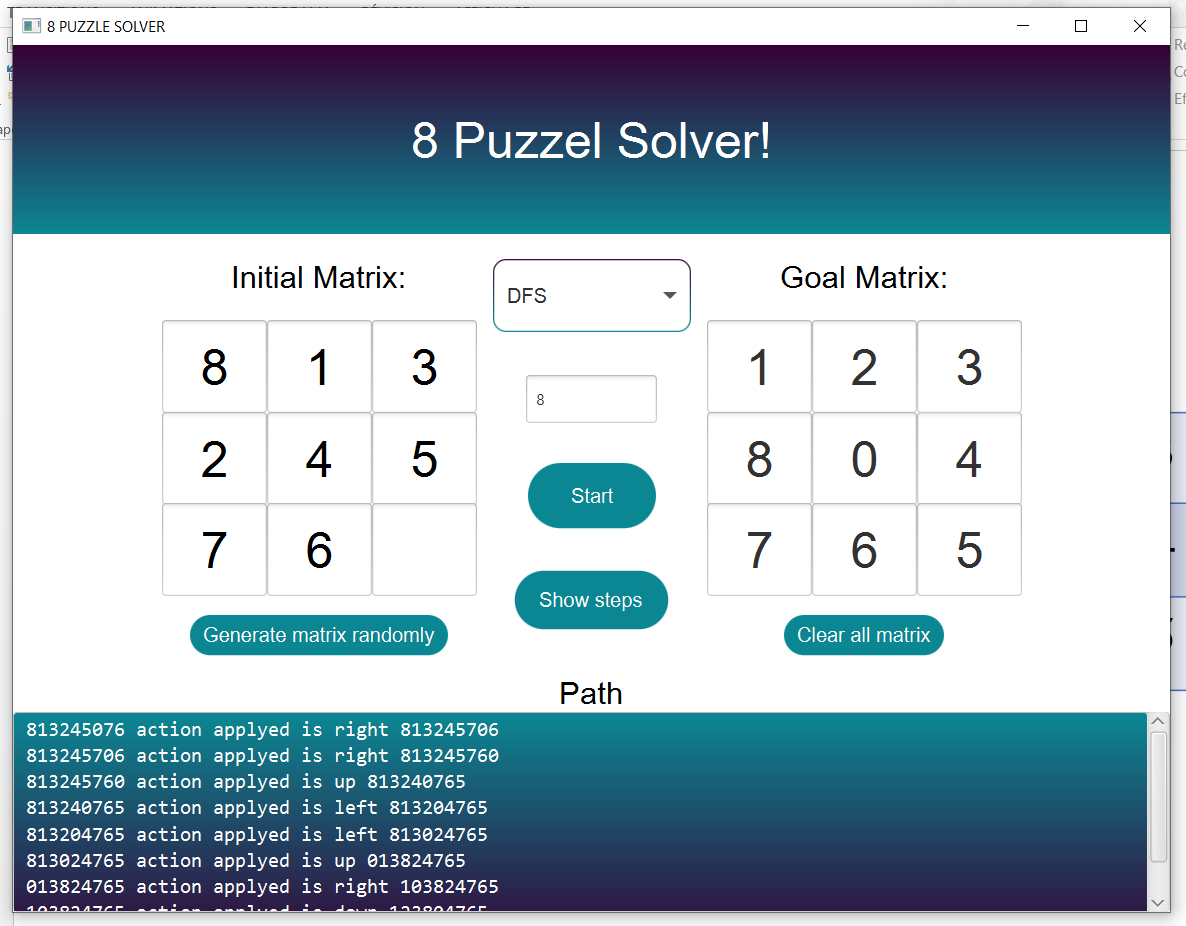
**Pour un Profondeur = 0 :**



**Pour un Profondeur = 5:**



**Pour un Profondeur = 8 :**



Déplacer la case vide à droit on obtient : 813245706

Déplacer la case vide à droit on obtient : 813245760

Déplacer la case vide en haut on obtient : 813240765

Déplacer la case vide à gauche on obtient : 813204765

Déplacer la case vide à gauche on obtient : 813024765

Déplacer la case vide en haut on obtient : 013824765

Déplacer la case vide à droit on obtient : 103824765

Déplacer la case vide en bas on obtient : 123804765

Le but est **atteint !**

Donc le chemin est : **right – right – up – left – left – up – right – down.**

### Complexité temporelle et spatiale :

**DFS sans limite de profondeur :**Consiste à parcourir le graphe en profondeur. Il s’agit d’aller à chaque fois à la plus basse profondeur avant de revenir explorer d’autres nœuds. Cette technique n’est pas optimale, ne garantit pas que le but trouvé est optimal. Sa complexité en temps est de O(b**m**) (b est le facteur de branchement et m est la profondeur maximale de l’arbre de recherche). En effet, au pire des cas l’algorithme crée et examine tous les nœuds du graphe.

Le Depth first search a une complexité en espace de **O(bm)**; elle offre une meilleure complexité en espace **O(mb)**lorsqu’il s’agit d’un tree-search ; en considérant qu’au pire des cas le but se retrouve à la profondeur maximale.

**DFS avec limite de profondeur:**

Cet algorithme vient pallier au problème de profondeur infinie du depth first search sans limite de profondeur. Ainsi le depth first search avec profondeur limité consiste à effectuer la recherche en profondeur (Depth First Search) en limitant la profondeur à atteindre lors du parcours de l’arbre de recherche. L’inconvénient principal de cette technique est le choix de la profondeur à fixer comme limite. En considérant ***L*** la limite choisie, Limited depth search n’est pas complet si la limite choisie est inférieure à la profondeur de la solution la moins coûteuse (L<d) (d est la profondeur de la solution la moins couteuse) et ne garantit pas l’optimalité si la limite choisie est supérieure à la profondeur de la solution la moins coûteuse L > d. Sa complexité en temps est **O (bL)** avec b est le facteur de branchement maximal. En effet, au pire des cas l’algorithme crée et examine tous les noeuds du graphe jusqu’à la limite L.

Sa complexité en espace **O (bL),** mais **O (bL)** lorsqu’il s’agit d’un tree-search ; en considérant qu’au pire des cas le but se retrouve à la profondeur maximale.

# Résolution avec L’algorithme A\* :

## Définition du l’algorithme A\* :

L'algorithme de recherche A \* est l'une des meilleurs techniques et des plus populaires utilisées dans la recherche de chemin et les traversées de graphes.

De manière informelle, les algorithmes de recherche A \*, contrairement aux autres techniques de parcours, ont un « cerveau ». Cela signifie qu'il s'agit d'un algorithme intelligent qui le distingue des autres algorithmes conventionnels.

Et il convient également de mentionner que de nombreux jeux et cartes Web utilisent cet algorithme pour trouver le chemin le plus court de manière très efficace (approximation).

## Principe de l’algorithme :

Prenant notre problème de taquin comme exemple :

Le jeu est composé de **n\*m−1** petits carreaux numérotés à partir de 1 qui glissent dans un cadre du format **n× 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 du taquin est le problème **le plus grand de son type** qui peut **être résolu complètement** (on peut trouver toutes les solutions existent d’un problème donné). Il est simplement défini mais le problème est **NP-difficile.** **(Reinefeld 1993).**

Le problème est **grand combinatoirement** est exigé une résolution **guidée** afin d’atteindre avant **d’épuiser les ressources (temps et mémoire).**

L’espace d’état est de taille **(n\*m)!/2** ce qui fait **181 440** pour la variante **3×3.**

Dans ce projet notre objectif est atteindre l’état cible **(si possible)** depuis la cellule de départ **le plus rapidement possible.** Ici, l'algorithme de **recherche A \*** vient à la rescousse.

Ce que fait l'algorithme de **recherche A \*,** c'est qu'à chaque étape, il sélectionne **le nœud en fonction** d'une fonction **'f'** qui est un paramètre égal à la somme de deux autres paramètres ‘**g'** et **'h'.** À chaque étape, il sélectionne **le nœud/cellule** ayant **la plus petite valeur** de **« f »** et **traite ce nœud/cellule.**

On définit

**‘g’**: la **profondeur**

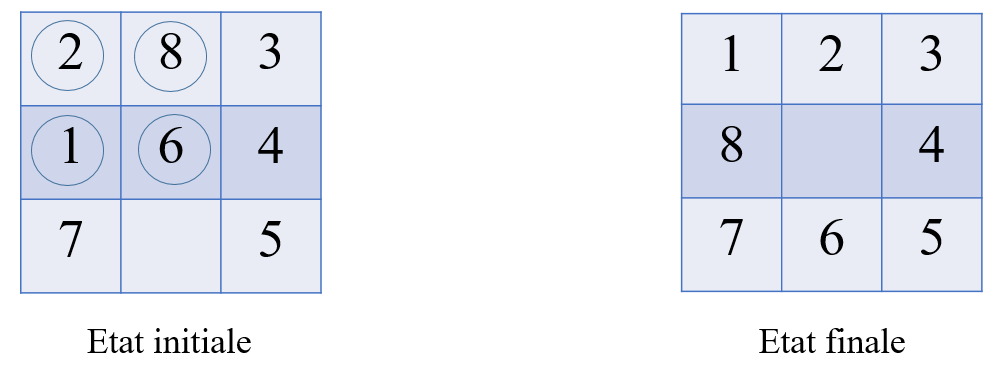
‘**h’** : l'**heuristique.**

Et on a la loi : **f(n) = g(n) + h(n)**

## Les heuristiques utiliser dans ce travail :

### Les cases mal placés :

Cette heuristique consiste à calculer le nombre de case mal placées entre le nœud courant et le nœud but

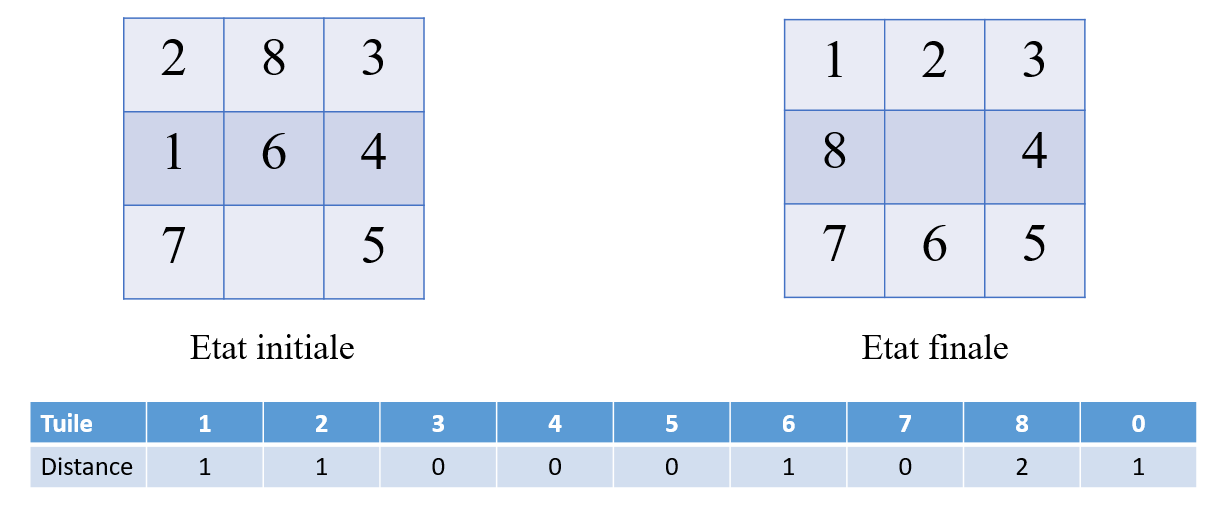


Nombre de case mal placées = h =5.

### La somme des distances de Manhattan :

Cette heuristique permet de sommes des distances de Manhattan des tuiles avec leur position à atteindre :

Exemple :



## La Complexité d’algorithme A\* :

Complexité temporelle :

* Dépend de l’heuristique.
* Dans le pire cas, il faut faudra visiter n nœuds …Mais, la taille de l’espace d’états (n) est généralement exponentielle à la taille du problème!

Complexité spatiale :

* O(n) où n est le nombre de nœuds explorés. Dans le pire cas n=|S|.

## Pseudo Code du l’algorithme A\* utilise :

**Entrée:** S: String *//contient l’état final;*

n : Node *//contient l’état, l’heuristique, la profondeur et le cout ;*

**Sortie :** Le chemin d’état initial jusqu’à but ;

**Var :** Antecedent *//contient (id, cout) de chaque nœud développer avec celle de son parent ;*

Open *// une file qui contient les nœuds à développer ;*

Actions *//Une file qui contient l’ensemble des actions pour atteindre l’état final ;*

Goal : booléen *// permet de sort de la boucle lorsque le but et atteindre ;*

nf : Node *//pour ne pas perdre le nœud initial ;*

Costs *//tableau ou ont sauvegarde les couts des nœuds développer ;*

**Début**

nf = n;

addToAntecedent(n);

Tantque (goal=false) faire

début

Expand (nf);

Si nœud (np) développer de np!=null

//le nœud np peut être après applique l’action up, down, left ou right

Alors addToAntecedent (np);

nf = getMinimalNode () ;

Si n.node = s

Alors goal=true ;

Fin ;

getPath (nf, n) ;

**Fin.**

## Les fonctions utilisées :

### Expand ():

**Entrée:** S : String *//contient l’état final;*

n : Node *//contient l’état, l’heuristique, la profondeur et le cout ;*

Costs *//tableau ou ont sauvegarde les couts des nœuds développer ;*

**Sortie :** quatre nœud développer *//left, right, up et down ;*

**Var :** n.left, n.right, n.up, n.down : Node *// les quatre nœuds à développer*

**Début**

n.left = left(n) ;

*//la fonction left permet de change la position de 0 dans le nœud avec son voisin gauche (la même chose pour right (), up () et down()).*

n.right = right(n) ;

n.up = up(n) ;

n.down = down(n) ;

Si les nœuds développer != null

Alors ajouter sans cout à la table **Costs ;**

**Fin.**

### AddToAntecedent() :

Cette fonction permet d’ajouter les nœuds développer avec leur couts a la table « Antecedent » pour but de nous aider à revenir en arrière pour avoir le chemin de but jusqu’à état initial (backtracking).

**Entrée :** n, np : Node *//n le nœud a ajouter au antécédent et np sont parent ;*

**Sortie :**

**Var :**

**Début :**

Si n existe dans Antecedent[][]

Alors si le cout de n < a le cout de nœud qui existe dans Antecedent [][]

Alors début

Ajouter n a Antecedent [][] ; OrganizeOpen(n) ;

Fin ;

**Fin.**

### OrganizeOpen () :

Cette fonction ne permet d’ajouter un nœud à la file open et l’organise en respectent la contraint suivant :

* Si nœud existe dans open *// on appelle nœud dans open « nopen »*

Alors si n.f < nopen.f

Alors ajouter n a open et supprimer nopen;

### GetMinimalNode ():

**Entrée :** Costs *//tableau ou ont sauvegarde les couts des nœuds développer ;*

Open *// une file qui contient les nœuds à développer ;*

**Sortie :** Nodemin : Node *//le nœud avec le cout minimal ;*

**Var :** min : entier ;

**Début**

min= (la table Costs trier) [0] *; //le premier élément de la table trier*

Nodemin= un nœud de open « nopen » tel que nopen.f=min ;

Retourner Nodemin ;

**Fin.**

### GetPath () :

Cette fonction nous permet de faire un retour arrière depuis le nœud but atteindre par la recherche vers l’état initial en utilisant la table « Antecedent ».

Ce traitement va aussi enregistre les actions affecter à ces nœud dans la table « Actions ».

L’inversement de cette table est le chemin d’aller d’état initial vers l’état but.

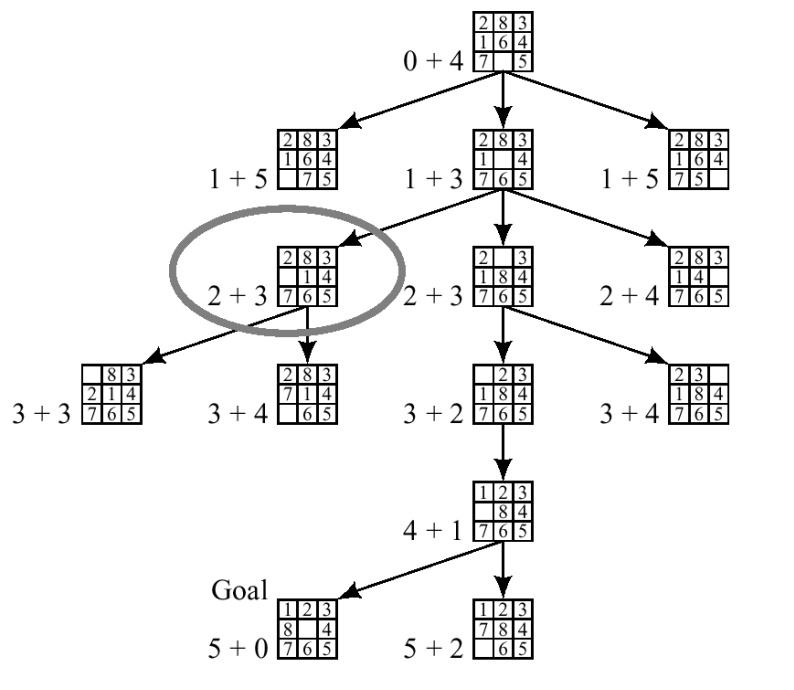
**Remarque :**

Tous parcours de la table « Antecedent » a était fait à l’aide de la fonction d’hachage suivante :

<Clé mod 362897>.

Exemple1 :

Avec l’heuristique **h1 « cases mal placées » :**

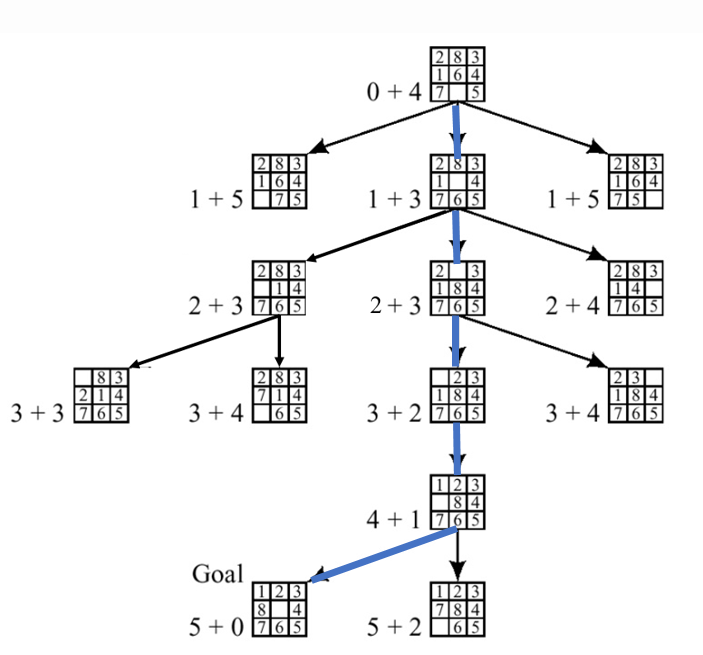
****

**Remarque :**

Dans le cas ou A\* trouve deux état avec le même cout, il choisit un état aléatoirement (voir l’état circuler a la figure si dessus).

Le but est obtenu en profondeur 5, mais on voit que le nombre de nœuds explores est moins que la recherche BFS et DFS !

Le chemin vers la solution selon l’algorithme A\* avec l’heuristique de cases mal placées sera :



Donc le chemin de l’état initial vers l’état final sera :

Déplacer la case vide en **HAUT**

Déplacer la case vide en **HAUT**

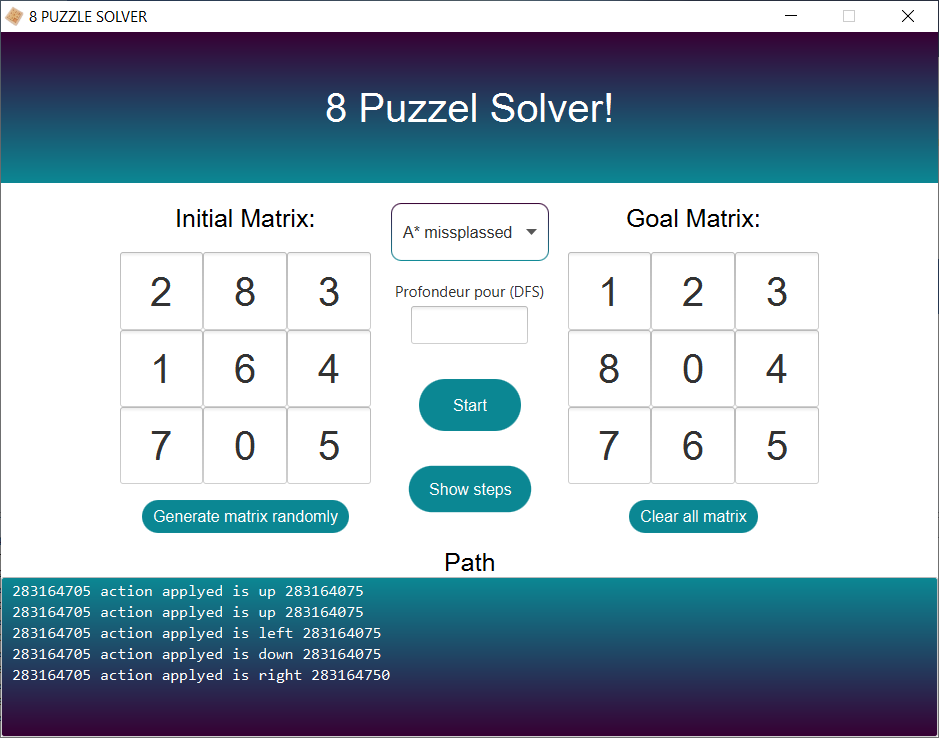
Déplacer la case vide vers la **GAUCHE**

Déplacer la case vide en **BAS**

Déplacer la case vide vers la **DROITE**

Donc le chemin est : **UP – UP – LEFT – DOWN – RIGHT**

Test de l’exemple précédant dans l’Application :



Déplacer la case vide en **HAUT** on obtient : 283104765

Déplacer la case vide en **HAUT** on obtient : 203184765

Déplacer la case vide à **GAUCHE** on obtient : 023184765

Déplacer la case vide en **BAS** on obtient : 123084765

Déplacer la case vide à **DROITE** on obtient : 123804765

Le but est **atteint !**

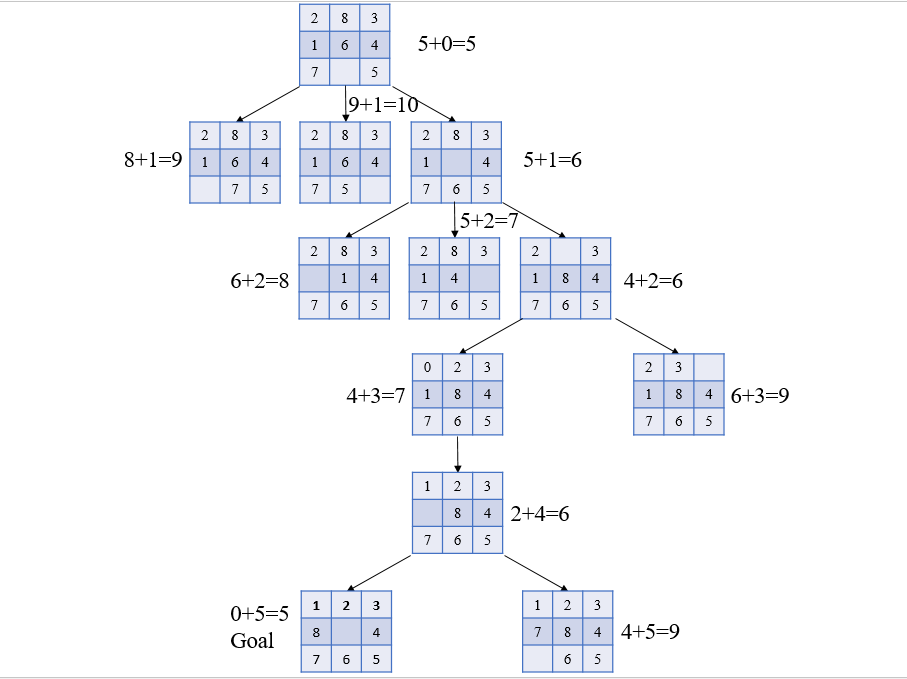
Le chemin obtenu par l’algorithme A\* avec l’heuristique de cases mal placées selon l’application est :

**UP – UP – LEFT – DOWN – RIGHT**

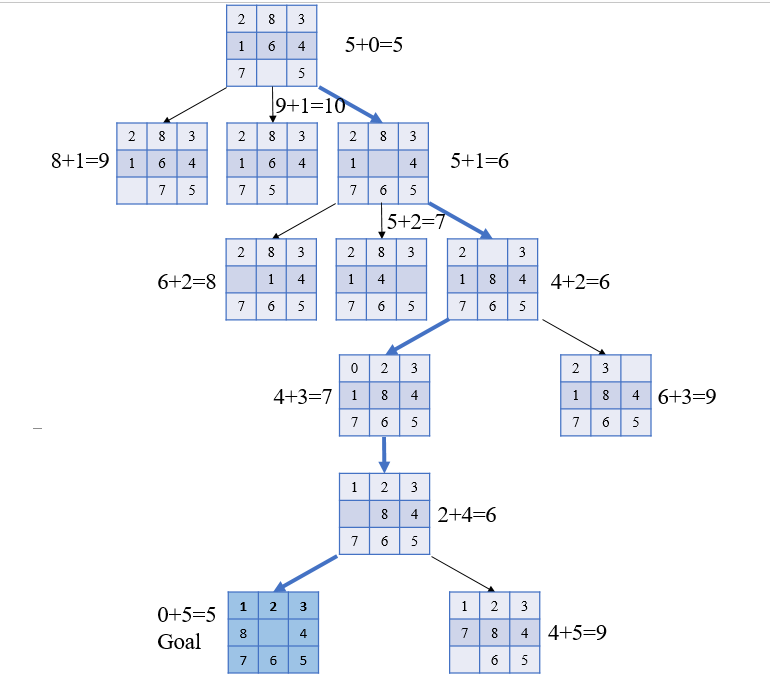
Ce qui est le même avec le chemin qu’on a obtenu manuellement.

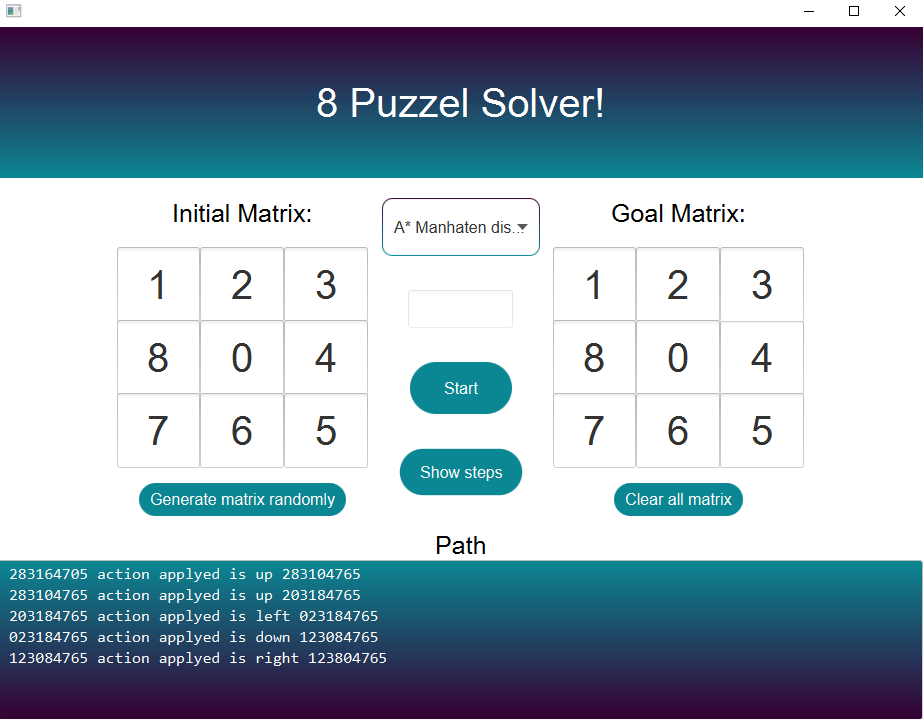
Exemple 2 :

Avec l’heuristique **h2 « distance de Manhattan » :**



Le chemin vers la solution selon l’algorithme A\* avec l’heuristique de distance de Manhattan sera :



 **Test de l’exemple précédant dans l’Application :**

Déplacer la case vide en HAUT on obtient : 283104765

Déplacer la case vide en HAUT on obtient : 203184765

Déplacer la case vide à GAUCHE on obtient : 023184765

Déplacer la case vide en BAS on obtient : 123084765

Déplacer la case vide à DROITE on obtient : 123804765

Le but est **atteint !**

Le chemin obtenu par l’algorithme A\* avec l’heuristique de distance de Manhattan selon l’application est :

**UP – UP – LEFT – DOWN – RIGHT**

**Comparaison des temps d’exécution**

Temps d’exécution en nanoseconde :

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Matice** | **NumMatrice** | **A\* misplaced(ns)** | **A\* manhaten distance(ns)** | **Dfs(ns)** | **Bfs(ns)** |
| 283164705 | 1 | 8848700 | 7031200 | 43025900 | 899300 |
| 813245706 | 2 | 17936200 | 12069500 | 61704200 | 2284000 |
| 013845276 | 3 | 27021900 | 18534000 | 53937000 | 4930600 |
| 675031842 | 4 | 1965736000 | 160499700 | 384048600 | 179094000 |
| 518603247 | 5 | 11815542600 | 1136714400 | 58217501 | 510319500 |

**Evaluation des méthodes de recherches :**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Methode | Complétude | Complexité temporelle | Complexité spatiale | Optimalité |
| BFS | Oui (si b fini ) | O(bd) | O(bd) | Oui |
| DFS | Oui ( si b fini et pas de cycle) | O(bm) | O(b\*m) | Non |
| A\* | Oui | O(bd) ( polynomiale si h optimale) | O(bd) | Oui |

Les compléxité temporelle et spatiale dépendent de :

b : Facteur de branchement maximum de l’arbre de recherche

d : Profondeur à laquelle se trouve le meilleur noeud-solution

m : Profondeur maximum de l’éspace de recherche

**Conclusion :**

**Analyse des résultats:**

Nous avons trouvé conformément à nos attentes que l’algorithme A\* est le mieux adapté à ce problème. Les autres algorithmes (BFS /DFS ) 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.