

PROJET INTELLIGENCE

ARTIFICIELLE

JEU DE TAQUIN 3X3

Travail par :



Ilyes Rezgui



Sarra Touati

INSTITUT
SUPERIEUR
INFORMATIQUE
المعهد العالي للإعلامية

Groupe :



L2CS4



Un Aperçu sur l'output de Notre jeu



Sommaire :

Chapitre 1 : Introduction au jeu

Chapitre 2 : Les Trois algorithmes de recherche qui seront appliqués : DFS, BFS, A *.

Chapitre3 : Implémentation des Algorithmes.

Chapitre4 : Comparaison entre les trois algorithmes.

Chapitre5 : Interface graphique (TKinter)

Chapitre N°1 : Introduction au Jeu

- Ce jeu, imaginé par Sam Loyd (1870), suscita immédiatement un grand intérêt à travers l'Occident. L'une des raisons peut-être de ce rapide succès était une récompense de mille dollars promis par Lloyd à quiconque parviendrait à transformer une position de départ en une position d'arrivée fixée
- Le jeu du taquin est un puzzle constituer de cases ayant une structure modifiable entre eux de sorte qu'on puisse les mélanger, le but étant de les remettre dans leur ordre d'origine.
- il est un puzzle coulissant 3x3 qui se compose d'un cadre de huit tuiles carrées numérotées dans un ordre aléatoire avec une tuile manquante
- On peut appliquer certain algorithme Pour faciliter la recherche de Solution , On va voir ça dans les chapitres suivants.

Chapitre 2 : Les Trois algorithmes de recherche qui seront appliqués : DFS, BFS, A *

Dans ce chapitre, nous parlerons de chaque algorithme mentionné

BFS (recherche en largeur) :

L'algorithme de **parcours en largeur** (ou BFS, pour Breadth-First Search en anglais) permet le parcours d'un graphe ou d'un arbre de la manière suivante : on commence par explorer un nœud source, puis ses successeurs, puis les successeurs non explorés des successeurs, etc.

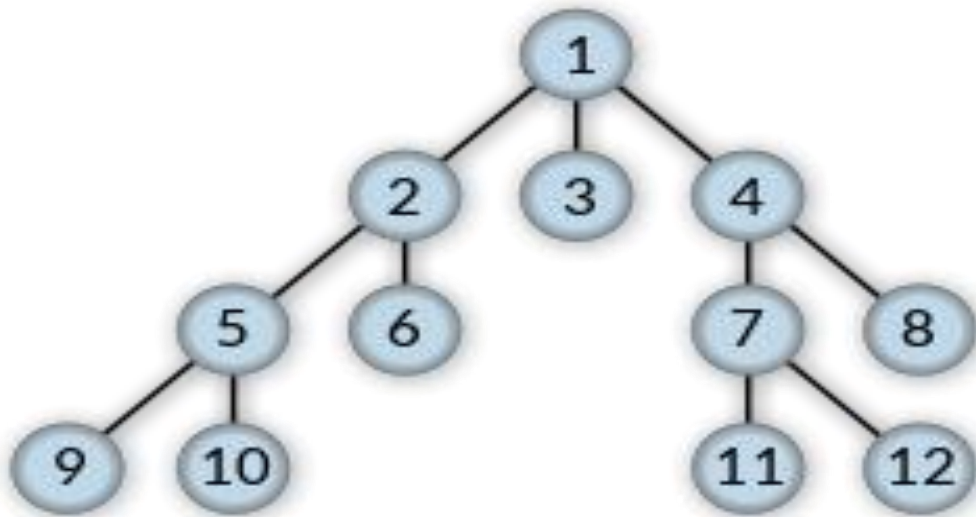
On va procéder à un parcours en largeur du graphe en mettant les sommets successifs dans une file (**structure FIFO**).

Voici la description intuitive de l'algorithme :

1. On enfile le sommet de départ (on visite la page d'accueil du site).
2. On enfile les sommets adjacents à la tête de file (on visite les pages ciblées par la page d'accueil) s'ils ne sont pas déjà présents dans la file.
3. On défile(c'est-à-dire on supprime la tête de file).

4. Tant que la file n'est pas vide, on ré-itére les points 2 et 3.

En d'autres termes, on défile toujours prioritairement les sommets (les pages) les plus tôt découverts.



Ordre de visite pour cet exemple :

1-2-3-4-5-6-7-8-9-10-11-12

Complexité de BFS = $O(V + E)$ où V est les sommets et E les arêtes

DFS (recherche en profondeur) :

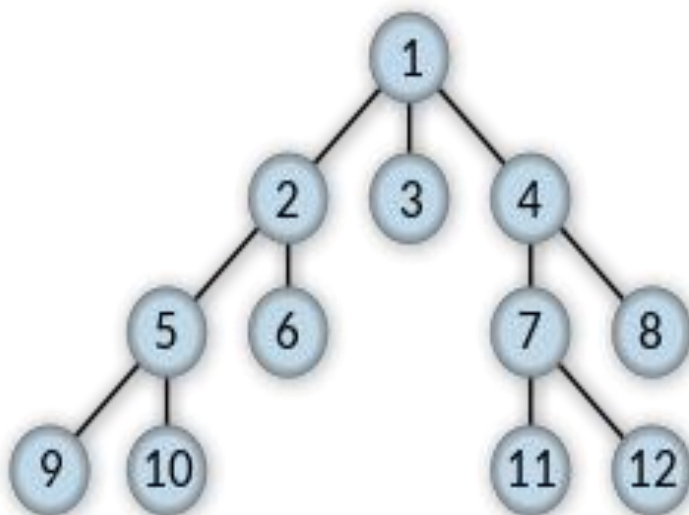
L'algorithme de parcours en profondeur (ou DFS, pour Depth-First Search en anglais) permet le parcours d'un graphe ou d'un arbre de la manière suivante : Il commence à explorer le graphique à partir d'un nœud et explore sa profondeur avant de revenir en arrière.

Une implémentation DFS standard place chaque sommet du graphe dans l'une des deux catégories suivantes:

- Visité
- Non visité

Une pile est conservée dans cet algorithme pour stocker les nœuds suspendus pendant la traversée.

A la fin de l'exécution de l'algorithme, tout les nœuds seront visités.



Ordre de visite
pour cet exemple :

1-2-5-9-10-6-3-4-
7-11-12-8

A* (A star algorithm) :

L'algorithme A* est un algorithme de recherche heuristique permettant de trouver le meilleur chemin d'un état initial à l'état final.

A chaque étape, il sélectionne le nœud en fonction de 'F' ayant la plus basse valeur.

Sachant que :

F :

et $F=G+H$

G : le coût de déplacement pour se déplacer du point de départ à une case donnée de la grille, en suivant le chemin généré pour s'y rendre.
(niveau de profondeur dans l'arbre)

H : le coût de déplacement estimé pour se déplacer de cette case donnée sur la grille à la destination finale (nombre de cases mal placées)

C'est ce qu'on appelle souvent l'heuristique, qui n'est rien d'autre qu'une sorte de supposition intelligente.

Chapitre N°3 : Implémentation des Algorithmes.

BFS :

```
File Edit Format Run Options Window Help
from copy import deepcopy
import timeit

etat_final = [[1, 2, 3],
              [8, 0, 4],
              [7, 6, 5]]
opérateurs_de_transformations = {"U" : [-1, 0], "D" : [1, 0], "L" : [0, -1], "R" : [0, 1]}
class Taquin :
    def __init__(self, matrice_courante, matrice_precedente, operation) :
        self.matrice_courante = matrice_courante
        self.matrice_precedente = matrice_precedente
        self.operation = operation

    def coordonnees(taquin, cellule) :
        for ligne in range(3) :
            if cellule in taquin[ligne] :
                return (ligne, taquin[ligne].index(cellule))

    def appliquer_operations(taquin, open, closed) :
        pos_vide = coordonnees(taquin.matrice_courante, 0)

        for operation in opérateurs_de_transformations:
            new_pos = ( pos_vide[0] + opérateurs_de_transformations[operation][0], pos_vide[1] + opérateurs_de_transformations[operation][1] )
            if 0 <= new_pos[0] < 3 and 0 <= new_pos[1] < 3 :
                new_matrix = deepcopy(taquin.matrice_courante)
                new_matrix[pos_vide[0]][pos_vide[1]] = taquin.matrice_courante[new_pos[0]][new_pos[1]]
                new_matrix[new_pos[0]][new_pos[1]] = 0
                if str(new_matrix) not in closed.keys() :
                    open[str(new_matrix)] = Taquin(new_matrix, taquin.matrice_courante, operation)

    def chemin_solution(closed_liste) :
        taquin = closed_liste[str(etat_final)]
        branche = list()
```

1 2 3 4 5

Activer Windows
Accédez aux paramètres pour activer Windows.

```
File Edit Format Run Options Window Help
while taquin.operation :
    branche.append({
        'operation' : taquin.operation,
        'taquin' : taquin.matrice_courante
    })
    taquin = closed_liste[str(taquin.matrice_precedente)]
    branche.append({
        'operation' : 'taquin initial sans opération de transformation',
        'taquin' : taquin.matrice_courante
    })
    branche.reverse()
    return branche

def main(puzzle_initial) :
    start_algo=timeit.default_timer()
    open_liste = {str(puzzle_initial) : Taquin(puzzle_initial, puzzle_initial, "")}

    closed_liste = {}
    while True :
        taquin_a_traiter = list(open_liste.values())[0]

        closed_liste[str(taquin_a_traiter.matrice_courante)] = taquin_a_traiter
        if taquin_a_traiter.matrice_courante == etat_final :
            stop_algo = timeit.default_timer()
            time = stop_algo-start_algo

            return chemin_solution(closed_liste), len(closed_liste), format(time, '.8f')

        appliquer_operations(taquin_a_traiter, open_liste, closed_liste)

        del open_liste[str(taquin_a_traiter.matrice_courante)]
```

6 7

Activer Windows
Accédez aux paramètres pour activer Windows.



Explication De l'algorithme

- 1) Définition de l'état final
- 2) Définition des transitions possible a faire: left, right, up and down
- 3) Définir un constructeur pour la classe taquin qui est caractérisé par une matrice courante et une matrice précédente ainsi de la transition appliqué pour transitionner de la matrice précédente a la matrice courante
- 4) Définition d'une fonction qui retourne les cordonnés d'une cellule sous la forme (y,x) ou (ligne, colonne)
- 5) Définition d'une fonction qui insère un nœud déjà explore dans la file closed et ce nouveau qui va être suivant dans une file open
- 6) Définition d'une fonction qui retourne le chemin pour arriver à la solution sous la forme d'une liste de dictionnaires contenant la transition appliquer et la matrice courante
- 7) Définition d'une fonction main qui prend en paramètres le taquin initial et applique l'algorithme en faisant appel au fonctions définîtes précédemment



Exécution De l'algorithme


Python 3.10.4 (tags/v3.10.4:9d38120, Mar 23 2022, 23:13:41) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.

```
>>> = RESTART: C:\Users\User\Desktop\ilyes et sarra ia\ilyes et sarra ia\MainWindow.py  
chemin-solution vers le BUT: se trouve dans un fichier .txt cree apres cette execution && vous pouvez parcourir ce chemin dans l'interface graphique  
cout du chemin-solution (steps): 3  
total de noeuds explores/developpes a la fin de l'execution:  
11 noeuds +1 noeud-initial  
temps d'execution: 0.00362470
```

Total des noeuds explorés à la fin de l'exécution: 11 noeuds +1 noeud-initial
Coût du chemin-solution(steps): 3
Temps d'exécution: 0.00362470

Ci-dessous est le noeud initial.
Parcourez le chemin-solution en cliquant sur le bouton 'Noeud suivant' (il se trouve aussi dans un fichier BFS_OutPut.txt)

Noeud suivant



BFS_OutPut - Bloc-notes

Fichier Edition Format Affichage Aide

Chemin-solution: input-> [{'operation': 'taquin initial sans opération de transformation', 'taquin': [[1, 2, 3], [8, 6, 0], [7, 5, 4]]}, {'operation': 'D', 'taquin': [[1, 2, 3], [8, 6, 4], [7, 5, 0]]}, {'operation': 'L', 'taquin': [[1, 2, 3], [8, 6, 4], [7, 0, 5]]}, {'operation': 'U', 'taquin': [[1, 2, 3], [8, 0, 4], [7, 6, 5]]}]

<-BUT

cout du chemin-solution (steps): 3
total de noeuds explores/developpes a la fin de l'execution: 12
temps d'execution: 0.00362470

Activer Windows
Accédez aux paramètres pour activer Windows.

Ln 9, Col 1 | 100% | Windows (CRLF) | ANSI

DFS :

```
from copy import deepcopy
import timeit

etat_final = [[1, 2, 3],
               [8, 0, 4],
               [7, 6, 5]]

operateurs_de_transformations = {"R" : [0, 1], "L" : [0, -1], "D" : [1, 0], "U" : [-1, 0]}

class Taquin :

    def __init__(self, matrice_courante, matrice_precedente, operation) :
        self.matrice_courante = matrice_courante
        self.matrice_precedente = matrice_precedente
        self.operation = operation

def coordonnees(taquin, cellule) :
    for ligne in range(3) :
        if cellule in taquin[ligne] :
            return (ligne, taquin[ligne].index(cellule))

def appliquer_operations(taquin, open, closed) :
    pos_vide = coordonnees(taquin.matrice_courante, 0)

    for operation in operateurs_de_transformations:
        new_pos = (pos_vide[0] + operateurs_de_transformations[operation][0],
                  pos_vide[1] + operateurs_de_transformations[operation][1])
        if 0 <= new_pos[0] < 3 and 0 <= new_pos[1] < 3 :
            new_matrix = deepcopy(taquin.matrice_courante)

            new_matrix[pos_vide[0]][pos_vide[1]] = taquin.matrice_courante[new_pos[0]][new_pos[1]]
            new_matrix[new_pos[0]][new_pos[1]] = 0

            if str(new_matrix) not in closed.keys() :

                open[str(new_matrix)] = Taquin(new_matrix, taquin.matrice_courante, operation)

def chemin_solution(
    closed_liste) :
    taquin = closed_liste[str(etat_final)]
    branche = list()

    while taquin.operation :
        branche.append({
            'operation' : taquin.operation,
            'taquin' : taquin.matrice_courante
        })
        taquin = closed_liste[str(taquin.matrice_precedente)]
        branche.append({
            'operation' : 'taquin initial sans opération de transformation',
            'taquin' : taquin.matrice_courante
        })
        branche.reverse()

    return branche

def main(puzzle_initial) :

    start_algo=timeit.default_timer()

    open_liste = {str(puzzle_initial) : Taquin(puzzle_initial, puzzle_initial, "")}

    closed_liste = {}

    while True :

        taquin_a_traiter = list(open_liste.values())[-1]

        closed_liste[
            str(taquin_a_traiter.matrice_courante)] = taquin_a_traiter
        if taquin_a_traiter.matrice_courante == etat_final :
            stop_algo=timeit.default_timer()
            time=stop_algo-start_algo

            return chemin_solution(closed_liste), len(closed_liste), format(time, '.8f')

        appliquer_operations(taquin_a_traiter, open_liste,
                             closed_liste)

        del open_liste[str(taquin_a_traiter.matrice_courante)]
```



Explication De l'algorithme

1. Définition de l'état final
2. Definition des transitions possible a faire left, right, up and down
3. Définir un constructeur pour la classe taquin qui est caractérisé par une matrice courante et une matrice précédente ainsi de la transition appliquée pour transitionner de la matrice précédente a la matrice courante
4. Définition d'une fonction qui retourne les cordonnés d'une cellule sous la forme (y,x) ou (ligne, colonne)
5. Définition d'une fonction qui insère un nœud déjà visité dans la file closed et ces successeurs qui vont être visités par la suite dans une pile open.
6. Définition d'une fonction qui retourne le chemin solution sous la forme d'un dictionnaire contenant la transition effectuée et la matrice courante
7. Définition d'une fonction main qui prend en paramètres le taquin initial et applique l'algorithme en faisant appel aux fonctions définîtes précédemment.



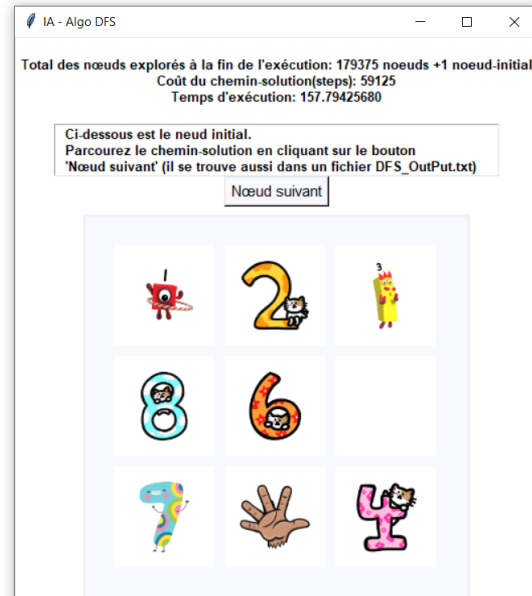
Exécution De l'algorithme

IDLE Shell 3.8.7

File Edit Shell Debug Options Window Help

Python 3.8.7 (tags/v3.8.7:6503f05, Dec 21 2020, 17:59:51) [MSC v.1928 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.

```
>>>
= RESTART: C:\Users\touat\AppData\Local\Programs\Python\Python38\ilYES et sarra ia\ilYES et sarra ia\ilYES et sarra ia\MainWindow.py
chemin-solution vers le BUT: se trouve dans un fichier .txt cree apres cette execution && vous pouvez parcourir ce chemin dans l'interface graphique
cout du chemin-solution (steps): 59125
total de noeuds explores/developpes a la fin de l'execution: 179375 noeuds +1 noeud-initial
temps d'execution: 157.79425680
```



DFS_OutPut - Notepad

File Edit Format View Help

```
Chemin-solution: input-> [{'operation': 'taquin initial sans opéra
6, 7, 3}], [1, 8, 4]]], {'operation': 'D', 'taquin': [[5, 7, 2], [6
': [[8, 1, 2], [0, 6, 3], [7, 5, 4]]], {'operation': 'R', 'taquin'
n': 'D', 'taquin': [[7, 2, 3], [5, 6, 0], [1, 8, 4]]], {'operation
8]]], {'operation': 'U', 'taquin': [[1, 0, 3], [6, 5, 4], [2, 7, 8
, 0, 4], [5, 1, 8]]], {'operation': 'U', 'taquin': [[7, 0, 3], [6,
: [[5, 3, 4], [1, 6, 8], [2, 7, 0]]], {'operation': 'L', 'taquin':
': 'L', 'taquin': [[0, 2, 4], [6, 1, 8], [3, 5, 7]]], {'operation'
]]], {'operation': 'L', 'taquin': [[0, 5, 4], [6, 3, 8], [1, 2, 7]
6, 7], [3, 0, 5]]], {'operation': 'U', 'taquin': [[1, 4, 8], [2, 0
[6, 3, 8], [0, 2, 7], [4, 1, 5]]], {'operation': 'D', 'taquin': [
'D', 'taquin': [[6, 1, 8], [0, 4, 7], [2, 3, 5]]], {'operation': '
, {'operation': 'U', 'taquin': [[2, 0, 7], [3, 8, 5], [4, 6, 1]]],
5], [0, 2, 1]]], {'operation': 'R', 'taquin': [[6, 4, 7], [8, 3, 5
, 2, 7], [3, 8, 5], [0, 4, 1]]], {'operation': 'R', 'taquin': [[6,
', 'taquin': [[0, 3, 5], [4, 7, 1], [8, 6, 2]]], {'operation': 'D'
{'operation': 'U', 'taquin': [[6, 8, 5], [7, 0, 1], [3, 4, 2]]], {
], [8, 0, 2]]], {'operation': 'U', 'taquin': [[6, 3, 5], [4, 0, 1]
4, 1], [0, 5, 2], [7, 6, 3]]], {'operation': 'D', 'taquin': [[8, 4
'taquin': [[6, 0, 1], [5, 7, 2], [4, 8, 3]]], {'operation': 'L', '
peration': 'U', 'taquin': [[6, 0, 1], [8, 4, 2], [7, 5, 3]]], {'op
[4, 1, 3]]], {'operation': 'U', 'taquin': [[7, 0, 2], [5, 8, 6], [
2], [8, 5, 6], [0, 7, 3]]], {'operation': 'R', 'taquin': [[1, 4, 2
aquin': [[7, 4, 2], [5, 8, 6], [0, 1, 3]]], {'operation': 'R', 'ta
ration': 'L', 'taquin': [[0, 5, 6], [1, 7, 3], [4, 2, 8]]], {'oper
, 0, 8]]], {'operation': 'U', 'taquin': [[2, 4, 6], [7, 0, 3], [5,
], [1, 7, 3], [2, 0, 8]]], {'operation': 'U', 'taquin': [[5, 4, 6]
quin': [[5, 0, 3], [2, 4, 8], [1, 6, 7]]], {'operation': 'L', 'taq
ation': 'R', 'taquin': [[6, 1, 3], [4, 2, 8], [5, 0, 7]]], {'opera
6, 7]]], {'operation': 'R', 'taquin': [[5, 1, 3], [2, 4, 8], [6, 0
[6, 5, 7], [1, 3, 4]]], {'operation': 'D', 'taquin': [[6, 2, 8], [
n': [[3, 1, 8], [5, 0, 7], [2, 6, 4]]], {'operation': 'U', 'taquin
on': 'U', 'taquin': [[2, 1, 8], [6, 0, 7], [3, 5, 4]]], {'operatio
5]]], {'operation': 'D', 'taquin': [[3, 6, 7], [1, 2, 4], [0, 8, 5]
```

Ln 1, Col 6146 100% Windows (CRLF) ANSI



A*

```
from copy import deepcopy
import timeit

etat_final = [[1, 2, 3], [8, 0, 4], [7, 6, 5]]
operateurs_de_transformations = {"U" : [-1, 0], "D" : [1, 0], "L" : [0, -1], "R" : [0, 1]}
class Taquin :

    def __init__(self, matrice_courante, matrice_precedente, g, h, operation) :
        self.matrice_courante = matrice_courante
        self.matrice_precedente = matrice_precedente
        self.g = g
        self.h = h
        self.operation = operation

    def f(self) :
        return self.g + self.h

def coordonnees(taquin, cellule) :
    for ligne in range(3) :
        if cellule in taquin[ligne] :
            return (ligne, taquin[ligne].index(cellule))

def cout_heuristique(etat_courant) :
    cout = 0
    for ligne in range(3) :
        for col in range(3) :
            if etat_courant[ligne][col] != 0:
                (ligne_final, col_final) = coordonnees(etat_final, etat_courant[ligne][col])
                cout += abs(ligne - ligne_final) + abs(col - col_final)

    return cout

def appliquer_operations(taquin, open, closed) :
    pos_vide = coordonnees(taquin.matrice_courante, 0)

    for operation in operateurs_de_transformations:
        new_pos = ( pos_vide[0] + operateurs_de_transformations[operation][0], pos_vide[1] + operateurs_de_transformations[operation][1] )
        if 0 <= new_pos[0] <3 and 0 <= new_pos[1] < 3 :
            new_matrix = deepcopy(taquin.matrice_courante)

            new_matrix[pos_vide[0]][pos_vide[1]] = taquin.matrice_courante[new_pos[0]][new_pos[1]]
            new_matrix[new_pos[0]][new_pos[1]] = 0

            g,h=taquin.g+1, cout_heuristique(new_matrix)
            if not ( str(new_matrix) in closed.keys() or \
                    str(new_matrix) in open.keys() and open[str(new_matrix)].f() < h+g ) :
                open[str(new_matrix)] = Taquin(new_matrix, taquin.matrice_courante,g,h, operation)
```

```
def meilleur_taqin(open_liste) :
    first_iter = True
    for taquin in open_liste.values() :
        if first_iter or taquin.f() < bestF :
            first_iter = False
            meilleur_taqin = taquin
            bestF = meilleur_taqin.f()
    return meilleur_taqin

def chemin_solution(closed_liste) :
    taquin = closed_liste[str(etat_final)]
    branche = list()
    while taquin.operation :
        branche.append({
            'operation' : taquin.operation,
            'taquin' : taquin.matrice_courante
        })
        taquin = closed_liste[str(taquin.matrice_precedente)]
    branche.append({
        'operation' : 'taquin initial sans opération de transformation',
        'taquin' : taquin.matrice_courante
    })
    branche.reverse()
    return branche

def main(puzzle_initial) :
    start_algo=timeit.default_timer()

    open_liste = {str(puzzle_initial) : Taquin(puzzle_initial, puzzle_initial, 0, cout_heuristique(puzzle_initial), "")}

    closed_liste = {}
    i=0
    while True :

        taquin_a_traiter = meilleur_taqin(open_liste)
        closed_liste[str(taquin_a_traiter.matrice_courante)] = taquin_a_traiter

        if taquin_a_traiter.matrice_courante == etat_final :
            stop_algo=timeit.default_timer()
            time=stop_algo-start_algo

            return chemin_solution(closed_liste), len(open_liste)-1+len(closed_liste), format(time, '.8f')

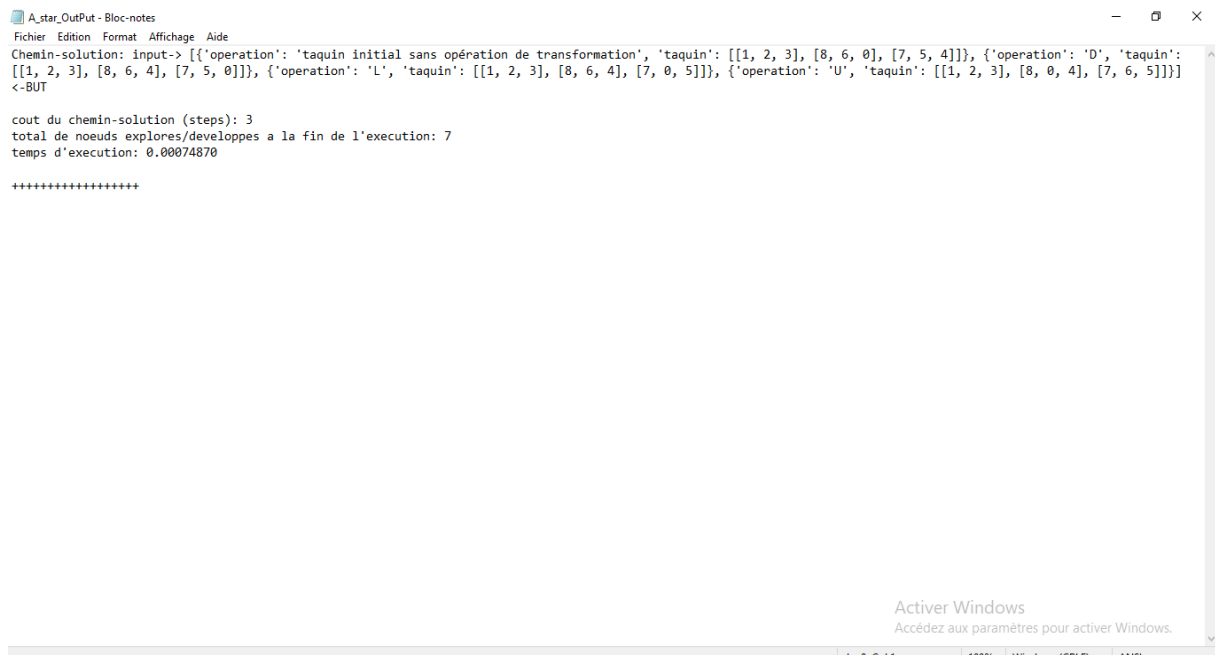
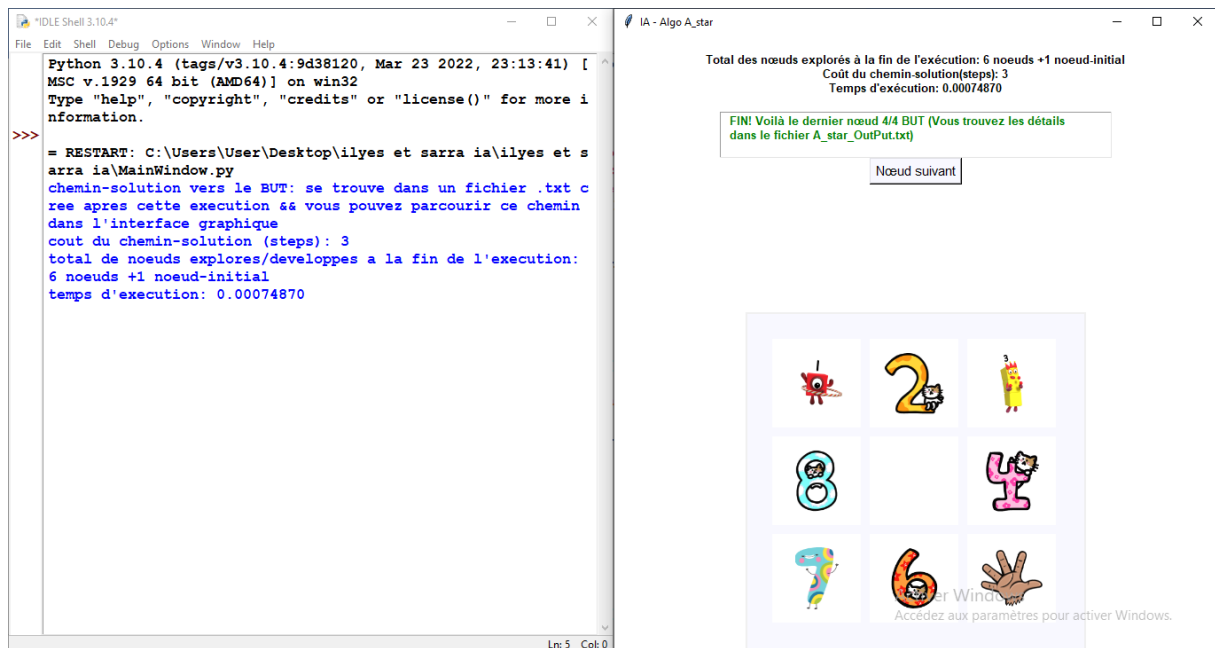
        appliquer_operations(taquin_a_traiter, open_liste, closed_liste)

        del open_liste[str(taquin_a_traiter.matrice_courante)]
```



Explication De l'algorithme

1. Définition de l'état final
2. Definition des transitions possible à faire left, right, up and down
3. Définir un constructeur pour la classe taquin qui est caractérisé par sa matrice de valeurs, la matrice du taquin antécédent, l'opération résultante et les valeurs de l'algorithme A^* , $F=G+H$.
4. Définir une fonction calculant le paramètre F
5. Définir une fonction qui retourne les coordonnées d'une cellule sous la forme (y,x) ou (ligne, colonne)
6. Définir une fonction calculant l'heuristique H (coût total du déplacement de chaque case de son état actuel à l'état final)
7. Définir une fonction qui insère les fils de t dans open après avoir appliqué toutes les opérations possibles {U,D,R,L}
8. Définir une fonction qui renvoie le meilleur taquin de l'ensemble des taquins fils -meilleur c.-à-d. plus petite $f(n)=g(n)+h(n)$
9. Définition d'une fonction qui retourne le chemin solution sous la forme d'un dictionnaire contenant la transition effectuée et la matrice courante
10. Définition d'une fonction main qui prend en paramètres le taquin initial et applique l'algorithme en faisant appel aux fonctions définîtes précédemment.



Chapitre N°4 : Comparaison entre les trois Algorithmes.

	DFS	BFS	A*
DFS		<ul style="list-style-type: none"> -Si l'arbre a un grand facteur de ramification (BFS devient lent avec des arbres larges) -Si des solutions et fréquentes et situées au fond de l'arbre -S'il y a une limite à la quantité de mémoire pouvant être utilisée 	<ul style="list-style-type: none"> Si l'heuristique de A* est mauvaise Si le but est l'optimalité et que l'heuristique A* n'est pas admissible Si les solutions sont fréquentes et situées au plus profond de l'arbre
BFS	<ul style="list-style-type: none"> Si au moins l'un des deux est requis : optimalité, exhaustivité -Si l'arbre est infini Si la profondeur maximale est beaucoup plus grande que le facteur de branchement 		<ul style="list-style-type: none"> Si l'arbre a un faible facteur de ramification Si l'arbre est dense Si l'heuristique est médiocre Si l'heuristique n'est pas admissible et que l'optimalité est requise

	<p>Si vous savez que la solution est maintenant loin de la racine de l'arbre</p> <p>Si les solutions sont rares et situées au plus profond de l'arbre</p> <p>Quand l'arbre est clairsemé (je ne sais pas pourquoi)</p>		
A*	<p>Si l'arbre est infini</p> <p>Si l'arbre est dense</p> <p>En général, la recherche aveugle est plus lente que la recherche heuristique, donc pour une heuristique assez bonne, A* doit être préféré</p>	<p>Si l'espace mémoire est limité</p> <p>Si l'arbre a un facteur de ramification élevé</p> <p>Si l'arbre est dense</p> <p>Bien que la complexité de la file d'attente soit légèrement meilleure que celle de la file d'attente prioritaire, la complexité temporelle de A* est</p>	

		généralement meilleure que la complexité temporelle de BFS avec une heuristique assez bonne	
--	--	--	--

Comparaison entre les trois algorithmes (Autres stratégies) :

Algorithme	Complet ?	Optimal ?	Complexité En temps	Complexite espcae
DFS	Non	Non	$O(b^m)$	$O(b^m)$
BFS	Oui	Oui si cout=1	$O(b^d)$	$O(b^d)$
A*	Oui	Oui si l'heuristique est \leq a la vérité	Nombre des nœuds avec $g(n)+h(n) \leq C^*$	

Chapitre N°5 : Interface Graphique (TKinter).

```
from Algorithms import A_star_algo, BFS_algo, DFS_algo
from Algorithms.A_star_algo import etat_final
import tkinter as tk
from random import shuffle

global n, branche_solution, etapes_solution, Nbr_total_noeuds_explores, temps

def melanger():
    global input_default
    L = list(i for j in input_default for i in j)
    shuffle(L)
    input_default= [[L[0],L[1],L[2]], [L[3],L[4],L[5]], [L[6],L[7],L[8]]]
    for i in range(9):
        listeT = list(i for j in input_default for i in j)
        ligne, col = i // 3, i % 3
        can.create_image(30 + 110 * col, 30 + 110 * ligne, anchor=tk.NW, image=img[listeT[i]])

def noeud_suivant():
    global n
    n+=1
    for i in range(9):
        if n>etapes_solution:
            break
        else:
            listeT = list(i for j in branche_solution[n]['taquin'] for i in j)
            ligne, col = i // 3, i % 3
            if n==etapes_solution:
                texte.delete("1.0", tk.END)
                texte.insert(tk.END, "FIN! Voilà le dernier noeud {}/{} BUT (Vous trouvez les détails dans le fichier {}_OutPut.txt)".f
                texte['fg']='green'
                texte.pack()
```

```
def DFS():
    solve('DFS')
def BFS():
    solve('BFS')
def A_star():
    solve('A_star')

def solve(algo):
    global branche_solution, etapes_solution, img, can2, texte, n, Nbr_total_noeuds_explores, algorithm
    n=0
    algorithm=algo
    if algo=='A_star':
        branche_solution, Nbr_total_noeuds_explores, temps = A_star_algo.main(input_default)
    elif algo=='BFS':
        branche_solution, Nbr_total_noeuds_explores, temps = BFS_algo.main(input_default)
    elif algo=='DFS':
        branche_solution, Nbr_total_noeuds_explores, temps=DFS_algo.main(input_default)

    etapes_solution = len(branche_solution) - 1

    print("chemin-solution vers le BUT: se trouve dans un fichier .txt cree apres cette execution ^^ vous pouvez parcourir ce chemin c
    print("cout du chemin-solution (steps): {}".format(etapes_solution))
    print("total de noeuds explores/developpes a la fin de l'execution: {} noeuds +1 noeud-initial".format(Nbr_total_noeuds_explores-1)
    print("temps d'execution: {}".format(temps))

    file = open(algo+'_OutPut.txt', 'a')
    file.write("Chemin-solution: input-> "+str(branche_solution)+" <-BUT\n\n")
    file.write("cout du chemin-solution (steps): "+str(etapes_solution)+"\n")
    file.write("total de noeuds explores/developpes a la fin de l'execution: "+str(Nbr_total_noeuds_explores)+"\n")
```

```

file.write("Chemin-solution: input-> "+str(branche_solution)+" <-BUT\n\n")
file.write("cout du chemin-solution (steps): "+str(etapes_solution)+"\n\n")
file.write("total de noeuds explores/developpes a la fin de l'execution: "+str(Nbr_total_noeuds_explores)+"\n\n")
file.write("temps d'execution: "+temps+"\n\n")
file.write("+++++\n\n")
file.close()

mainWindow.destroy()

AstarWindow = tk.Tk()
AstarWindow['bg'] = 'white'
AstarWindow.title('IA - Algo {}'.format(algo))
AstarWindow.geometry("520x560-400-150")

can2 = tk.Canvas(height=380, width=380, bg='ghost white')
can2.pack(side=tk.BOTTOM)
can2.pack()

img = []
for i in range(9):
    img.append(tk.PhotoImage(file="img/" + str(i) + ".png"))

tk.Label(text="Total des noeuds explorés à la fin de l'exécution: {} noeuds +1 noeud-initial\nCoût du chemin-solution(steps): {}\nNbr_total_noeuds_explores-1,etapes_solution, temps",bg='white',width=70,height=5,font=("Arial",10,"bold")).pack()

texte=tk.Text(font=("Arial", 10,"bold"),fg='black',bg='white',border=1,padx=10,width=60,height=3)
texte.insert(tk.END, "Ci-dessous est le neud initial.\nParcourez le chemin-solution en cliquant sur le bouton\n'Neud suivant' (il
texte.pack()
tk.Button(text='Neud suivant', font=("Arial", 11), fg='black',bg='ghost white', relief=tk.RAISED,
command=noeud_suivant).pack()

```

```

AstarWindow.mainloop()
mainWindow = tk.Tk()
mainWindow['bg'] = 'white'
mainWindow.title('IA - jeu de taquin')
mainWindow.geometry("440x500-400-150")
tk.Label(text="Choisissez l'algorithme pour résoudre dans le menu ci-dessus.\nVous pouvez également mélanger le puzzle.",bg='white',wi

can=tk.Canvas(height=380, width=380, bg='ghost white')
can.pack(side=tk.TOP)

img=[]
for i in range(9):
    img.append(tk.PhotoImage(file="img/"+str(i)+".png"))

input_default = [[1, 2, 3],
                 [8, 6, 0],
                 [7, 5, 4]]

for i in range(9):
    ListeT = list(i for j in input_default for i in j)
    ligne, col = i // 3, i % 3
    afficher = can.create_image(30 + 110 * col, 30 + 110 * ligne, anchor=tk.NW, image=img[ListeT[i]])

menu = tk.Menu(mainWindow)
menu.add_command(label="Mélanger", command=melanger)
menu.add_command(label="Résolution DFS", command=DFS)
menu.add_command(label="Résolution BFS", command=BFS)
menu.add_command(label="Résolution A*", command=A_star)
mainWindow.config(menu=menu)

mainWindow.mainloop()

```



Explication De l'algorithme

- 1) Définir une fonction mélanger qui permette de mélanger le taquin aléatoirement en faisant appel au module shuffle sur la liste l contenant toutes les cellules
- 2) Définition d'une fonction nœud suivant qui va être utiliser pour créer l'interface graphique liées aux emplacement des images des qu'on choisit l'algorithme à traiter
- 3) Définition des fonctions chaque une porte le nom de l'algorithme a appliquer ces fonctions vont être appelées lors l'appuie sur les boutons
- 4) Définition d'une fonction solve qui va calculer les paramètres demandées (chemin solution, cout chemin, temps d'exécution ...) leurs afficher sur le terminal et leurs stockées dans les fichier relatifs a l'exécution de chaque algorithme.
Dans cette étape aussi le traitement du passage d'un nœud a un autre va être fait en faisant appel à la fonction nœud suivant définie précédemment.
- 5) Création de l'interface graphique liée au main
- 6) Créer la canva ou les images vont être importées

- 7) Importer les images en utilisant le module photoimage de tkinter
- 8) Initialiser le taquin dans la zone de canva
- 9) Préparer le menu des commandes et faire appel aux fonction définies précédemment de chaque algorithme

Merci de votre attention

ANNEXE:

- # <https://fr.wikipedia.org/wiki/Taquin>
- # <https://www.geeksforgeeks.org/a-search-algorithm/>
- # <https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/>
- # <https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/>
- # https://www.youtube.com/watch?v=Vh6_Bf19W7E
- # <https://www.youtube.com/watch?v=N4M4W7JPOL4>
- # <https://www.youtube.com/watch?v=vP5TkF0xJgI>
- # <https://www.youtube.com/watch?v=eVsCO71q1L0>
- # <https://www.youtube.com/watch?v=oDqjPvD54Ss>