Rapport Projet IA :

Jeu du taquin

Présenté par :

ZANGAR Malak

&

MEKNI Maram

L2CS02

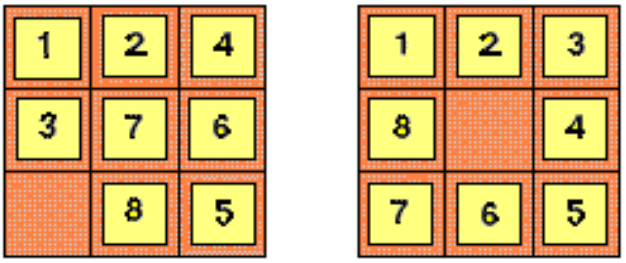
-2020/2021-

- Plan -

I/Chapitre 1 : Introduction :

Vous connaissez sans doute le jeu de taquin, constitué d'un cadre de 3× 3 cases, contenant 8 plaques et un trou qui permet de déplacer les plaques en glissant une à la place du trou.
Numérotons les plaques dans la position de départ du taquin de manière non standard.

La question est : quelles sont les positions du jeu de taquin que l'on peut obtenir par des mouvements autorisés à partir de la position initiale ?



Etat initial Etat but

Durant ce projet, nous avons réalisé un programme écrit en Python pour la résolution du taquin 3*3 partants de l'utilisation de différents algorithmes de recherche qui sont la recherche en profondeur, en largeur et A Etoile.

Nous avons choisi comme outils google collab vu qu'il est pratique, clair et organisé.

II/ Chapitre 2 : Stratégie de résolution :

Nous décrirons dans cette première partie le raisonnement suivit pour la réalisation du programme. Pour cela, nous partirons d'un taquin initial à corriger, par exemple : [1,2,3,8,6,0,7,5,4]

Le taquin que l'on veut obtenir aura la forme : [1,2,3,8,0,4,7,6,5]. La case contenant le numéro 0 représentera la case vide c'est à dire celle à déplacer. L'algorithme cherchera d'abord tous les déplacements possibles partant du taquin initial, et les mettra dans une liste d´états à visiter.

Dans notre programme :

- On a importé la fonction « deepcopy » de la bibliothèque « copy » qui copie de manière récursive un objet dans un autre objet, ça veut dire construire un nouvel objet composé puis, récursivement, insérer dedans des *copies* des objets trouvés dans l'objet original.
  Et la bibliothèque « time » qui permet de récupérer de temps spécifiques.

```
from copy import deepcopy
import time
```

- On a défini les 2 états initiale et finale.

```
t = [[1,2,3],[8,6,0],[7,5,4]]   #etat initiale
tf =[[1,2,3],[8,0,4],[7,6,5]]   #etat finale
```

- On a défini une fonction qui détecte l'état finale « estEtatFinal » et une fonction retournant la position du case vide.

```
def estEtatFinal (t,tf):
  return t==tf

def position_case_vide(t):
    for row in range(len(t)):
        for col in range(len(t[row])):
            if t[row][col] == 0 :
                return (row,col)
```

- On a défini une fonction qui retourne le contenu de la case ayant les coordonnées x et y passés en paramètres.

```
def numero(t,x,y):
  return t[x][y]
```

- On a défini la fonction « permuter » qui permet de permuter 2 cases adjacentes.

```
def permuter(t, pos, move):
    temp = deepcopy(t)
    i, j = pos  # ancienne position de vide
    x, y = move  # nouvelle position de vide
    temp[i][j], temp[x][y] = temp[x][y], temp[i][j]
    return temp
```

- On a défini la fonction « afficher » qui permet de représenter clairement notre taquin.

```
def afficher(t):
  print()
  for i in range(len(t)):
    for j in range(len(t[i])):
      print('+-----',end='')
    print('+')
    for j in range(len(t[i])):
      if t[i][j] != 0:
        print('| ',t[i][j],' ',end='')
      else:
        print('|     ',end='')
    print('|')
  for j in range(len(t[i])):
    print('+-----',end='')
  print('+')
```

Voilà l'aspect retourné :

```
+-----+-----+-----+
|  1  |  2  |  3  |
+-----+-----+-----+
|  8  |  6  |     |
+-----+-----+-----+
|  7  |  5  |  4  |
+-----+-----+-----+
```

- On a défini la fonction « valid » qui retourne les coordonnées possibles d'une case.

```
def valid(x,y):
    return x>-1 and x<3 and y>-1 and y<3
```

- On a défini la fonction « transition » qui retourne toutes les nouvelles matrices qu'on peut obtenir en partant de la matrice initiale et déplaçant la case vide dans ses positions possibles.

```
def transitions(t) :
  pos=position_case_vide(t)
  liste = []
  x = [1,0,-1,0]
  y = [0,1,0,-1]
  for i in range(4):
    if(valid(pos[0]+x[i],pos[1]+y[i])):
       liste.append([pos[0]+x[i],pos[1]+y[i]])
  derive=[]
  for i in liste :
    derive.append(permuter(t,pos,i))
  return derive
```

- On a défini la fonction heuristique « h » qui retourne le nombre des cases mal placés sans tenir compte du case vide.

```
def h(t,tf):
  nb=0
  for i in range(len(t)):
    for j in range(len(t[i])):
      if (t[i][j]!=tf[i][j])and (t[i][j]!=0):
        nb=nb+1
  return nb
```

III/Chapitre 3 :

Les algorithmes de résolution :

1. ***Recherche en largeur d'abord.***
2. ***Recherche en profondeur d'abord.***
3. ***Recherche A*.***

1. Recherche en largeur d'abord :

Un parcours en largeur débute à partir d'un nœud source. Puis il liste tous les voisins de la source, ensuite il les explore un par un. Ce mode de fonctionnement utilise donc une file dans laquelle il prend le premier sommet et place en dernier ses voisins non encore explorés. Les nœuds déjà visités sont marqués afin d'éviter qu'un même nœud soit exploré plusieurs fois. Voici les étapes de l'algorithme :

1. Mettre le nœud source dans la file.
2. Retirer le nœud du début de la file pour le traiter.
3. Mettre tous ses voisins non explorés dans la file (à la fin).
4. Si la file n'est pas vide reprendre à l'étape 2.

Implémentation de la fonction de recherche en largeur :

```
def bfs(t, tf):
  global start
  start=time.time()
  visited=[]
  queue=[]
  queue.append(t)
  visited.append(t)
  trace=[]
  while queue:
    f=queue.pop(0)

    if estEtatFinal(f,tf):
      trace.append(f)
      success=True
      break
    trace.append(f)
    trs=transitions(f)
    for j in trs:
     if j not in visited:
      visited.append(j)
      queue.append(j)
  return trace
```

La variable ***start*** récupère le temps de lancement de l'exécution. ***Visited*** est une liste qui contiendra les états visités pour ne les reparcourir plus. ***Queue*** est une liste qui contiendra les états à parcourir. ***Trace*** est une liste qui contiendra les états parcourus partant de l'état initial à l'état final. Elles sont toutes initialisés à vide.

On ajoute l'état initial à ***queue*** et ***visited.*** Tant que ***queue*** n'est pas vide, ***f*** recoit le 1er élément de ***queue*** et l'élimine de cette liste. On teste si ***f*** coïncide avec l'état final ou non. Si c'est le cas, on ajoute ***f*** à ***trace*** et ***success*** devient true , On quitte le tant que . Sinon on ajoute ***f*** à ***trace***, ***trs*** recoit les transitions possibles de ***f***. On parcours ces transitions l'une après l'autre, s'elles ne sont pas dans ***visited***, on les ajoute dans ***queue*** et dans ***visited*** . On reprend jusqu'à ***queue*** devient vide. Enfin, la fonction renvoi trace.

2. Recherche en profondeur d'abord :

L'exploration d'un parcours en profondeur depuis un sommet S fonctionne comme suit. Il poursuit un chemin dans le graphe jusqu'à un cul-de-sac ou jusqu'à atteindre un sommet déjà visité. Il revient alors sur le dernier sommet où on pouvait suivre un autre chemin puis explore un autre chemin. L'exploration s'arrête quand tous les sommets depuis S ont été visités. Bref, l'exploration progresse à partir d'un sommet S en s'appelant récursivement pour chaque sommet voisin de S.

Si G n'était pas un arbre, l'algorithme pourrait a priori tourner indéfiniment si on continuait l'exploration depuis un sommet déjà visité. Pour éviter cela, on marque les sommets que l'on visite, de façon à ne pas les explorer à nouveau.

L'implémentation de la méthode de recherche en profondeur :

```
trace= []
visited= []
success= False
def dfs(t, tf):
  global start
  start= time.time()
  global success
  if (success==False and t not in visited):
    trace.append(t)
    if estEtatFinal(t,tf):
      success=True
    visited.append(t)
    tab=transitions(t)
    for i in tab:
      if i not in visited and success==False:
        dfs(i,tf)
```

Trace contient les états parcourus par le taquin de l'état initial à l'état final. ***Visited*** contient les états déjà visités pour ne pas les revisiter . Elles sont initialisées à liste vide. ***Success*** initialisée à false donne true si l'état final est atteind. La variable ***start*** récupère le temps de lancement de l'exécution.

Si l'état actuel n'est pas encore visité et on n'a pas atteint l'état but on ajoute cet état à ***trace***. Si c'est l'état final, ***success*** devient True. Sinon elle reste à false. On ajoute l'état actuel à ***visited***. La variable ***tab*** contient les transitions possibles à partir de l'état actuel. On parcourt chaque élément de ***tab***,

s'il n'est pas visité déjà et si on n'a pas encore atteint l'état final, on reprend la fonction dés le début avec les nouveaux paramètres : la 1ere transition possible et l'état final jusqu'à atteindre l'état but. C'est le concept de la récursivité.

On appelle après la fonction dfs pour s'exécuter et on affiche tous les états parcourus dans ***trace*** et leur niveau.

3. Recherche A :*

Très utilisée en informatique, plus précisément en intelligence artificielle, en raison de sa simplicité elle est souvent présentée comme exemple typique d'algorithme de planification et elle est célèbre dans des applications comme les jeux vidéo privilégiant la vitesse de calcul sur l'exactitude des résultats.

L'algorithme A* est un algorithme de recherche de chemin entre un nœud initial et un nœud final. Il utilise une évaluation heuristique sur chaque nœud pour estimer le meilleur chemin y passant, et visite ensuite les nœuds par ordre de cette évaluation heuristique. C'est un algorithme simple, ne nécessitant pas de prétraitement, et ne consommant que peu de mémoire.

NB :

list.sort() a un paramètre *key* afin de spécifier une fonction qui peut être appelée sur chaque élément de la liste avant d'effectuer des comparaisons.
La valeur du paramètre *key* doit être une fonction qui prend un seul argument et renvoie une clef à utiliser à des fins de tri. Cette technique est rapide car la fonction clef est appelée exactement une seule fois pour chaque enregistrement en entrée.

Implémentation de la méthode de recherche A étoile :

```
def aetoile(t,tf):
  success=False
  start = time.time()
  niveaux = 0
  free_nodes = []
  free_nodes.append(t)
  closed_nodes = []
  while (free_nodes!=[]) and (not success) and (niveaux < 100):
    first_node = free_nodes[0]
    print()
    print ('** pas', niveaux,': **')
    niveaux += 1
    afficher(first_node)
    free_nodes.remove(first_node)
    closed_nodes.append(first_node)
    generated_states = transitions(first_node)

    for s in generated_states:
      if s == tf:
        success = True
        print (' \n** pas', niveaux,': **')
        afficher(s)
        goal_node = s
    free_nodes = free_nodes + generated_states
    free_nodes.sort(key = lambda el:(niveaux+h(el,tf)))
```

La variable ***success*** initialisée à false indique si on a atteint l'état but ou non. ***Start*** repère le début de temps d'exécution. ***Niveaux*** indique le niveau de cet état (le nombre de transitions faites à partir de l'état initial). ***free_nodes*** contient les nœuds à visiter. ***Closed_nodes*** contiendra les éléments déjà visités.

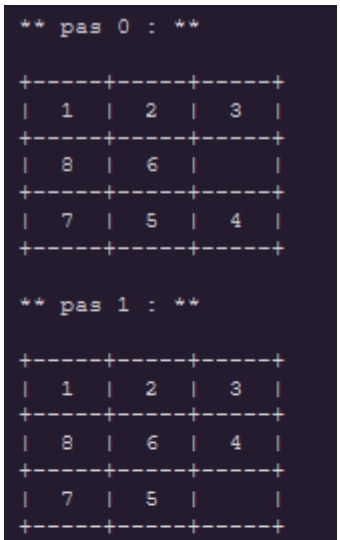
Tant que ***free_nodes*** n'est pas vide et on n'a pas atteint l'état but, ***first_node*** contiendra le 1er élément de ***free_nodes***, l'élémine de cette liste et l'ajoute à ***closed_nodes***. Ici, ***generated_states*** prend les états possibles à générer à partir de l'état actuel jusqu'à l'état but. On parcourt cette nouvelle liste élément par élément, si l'un est l'état final, ***success*** devient true et ***goal_node*** recoit l'état actuel. Puis on ajoute les ***generated_states*** à ***free_nodes*** et on les ordonne selon leur somme f(x) de : niveau actuel qui représente g(x) et le nombre de cases mal placés qui représente h(x).

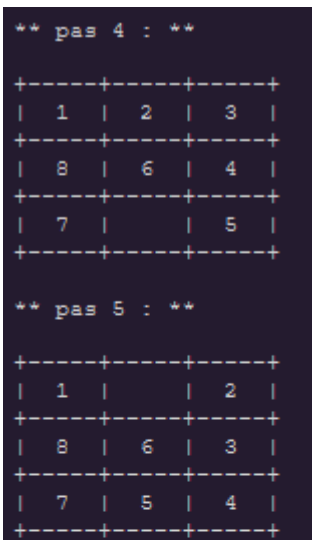
IV/ Chapitre 4 : Test :

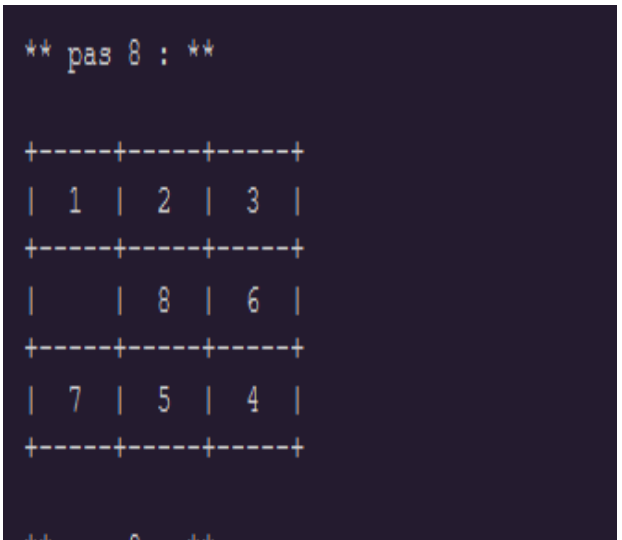
Le 1er test avec la recherche en largeur :

```
niveaux=0
trace=bfs(t, tf)
for i in trace :
    print()
    print ('** pas', niveaux,': **')
    afficher(i)
    niveaux+=1
print("\n --> Goal trouvé aprés",niveaux-1," iterations . \n")
print ('Time spent: %0.2fs' % (time.time()-start))
```

Résultat :







Le 2eme test avec la recherche en profondeur :

```
dfs(t,tf)
niveaux=0
for i in trace :
  print()
  print ('** pas', niveaux,' : **')
  afficher(i)
  niveaux+=1
print("\n --> Goal trouvé aprés",niveaux-1," iterations .")
print ('Time spent: %0.2fs' % (time.time()-start))
```

Résultat :

```
** pas 0  : **

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

** pas 1  : **

+-----+-----+-----+
|  1  |  2  |  3  |
+-----+-----+-----+
|  8  |  6  |  4  |
+-----+-----+-----+
|  7  |  5  |     |
+-----+-----+-----+
```

```
** pas 3  : **

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

 --> Goal trouvé aprés 3  iterations .
Time spent: 0.02s
```

Le 3eme test avec la recherche A :*

```
print (h(t,tf),"cases mal placées depuis l'etat initial")
aetoile(t,tf)
```

Résultat :

```
3 cases mal placées depuis l'etat initial

** pas 0 : **

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

** pas 1 : **

+-----+-----+-----+
|  1  |  2  |  3  |
+-----+-----+-----+
|  8  |  6  |  4  |
+-----+-----+-----+
|  7  |  5  |     |
+-----+-----+-----+
```

```
** pas 2 : **

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

** pas 3 : **

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

 --> Goal trouvé aprés 3  iterations .
Time spent: 0.01s
```

V/ Chapitre 5 :

Comparaison et Conclusion :

*** la recherche en largeur VS la recherche en profondeur :*

L'algorithme de parcours en profondeur diffère du parcours en largeur par le fait qu'il continue l'exploration jusqu'à arriver un cul-de-sac. C'est seulement à ce moment-là qu'il revient en arrière pour

explorer depuis un autre nœud voisin. L'utilisation d'une pile au lieu d'une file transforme l'algorithme du parcours en largeur en l'algorithme de parcours en profondeur.

	largeur	***A****
Profondeur	Les 2 algorithmes suivent une stratégie non informée, mais dans cet exemple la recherche en profondeur est plus performante que celle en largeur, en effet la 1ere dure 0.02s et atteint le but en 3 itérations, alors que la deuxième dure 0.06s et atteint le but en 9 itérations.	L'algorithme A* suit une stratégie informée qui lui donne un aspect de performance et rapidité, il dure 0.01s, l'algorithme de recherche en profondeur est semblable dans ce cas à A* en atteignant le but en 3 itérations seulement car c'est un exemple simple mais il reste toujours moins performant.

	Profondeur d'abord	Largeur d'abord	A*
Cout uniforme	X	X	
Cout variable			X
Sans heuristique	X	X	
Avec heuristique			X
optimalité	Non optimale	Non optimale en général Optimale si cout=1	
complétude	Non complète si espace d'états infini.	Complète si b est fini	
Complexité en temps	O(b^m)	O(b^d)	
Complexité en espace	O(b×m)	O(b^d)	
Structure de données	pile	file	

- Recherche en profondeur d'abord : utilise peu de mémoire, mais peut rater la solution optimale.
- Recherche en largeur d'abord : gourmande en mémoire, mais trouve toujours la solution optimale.
- Recherche heuristique : A* peut trouver la solution optimale de manière efficace.

Selon le cas testé dans notre projet, on peut constater que A* était la plus efficace.