Résolution de
problèmes de recherche

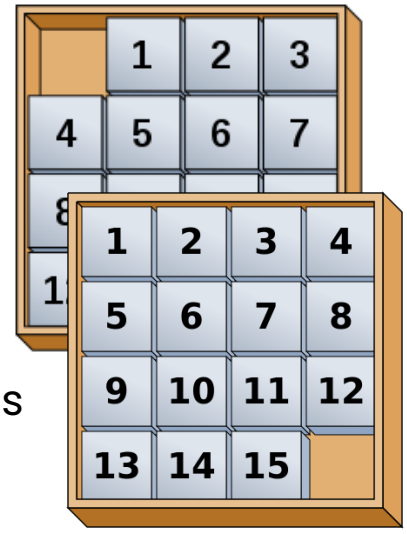
# Jeu de taquin

Réalisé par :
Eya Rabah
Ameni Belhadj

# PLAN :

# 01. Présentation de jeu de taquin

## Historique :




Tout d'abord, laissez-moi vous présenter le Taquin. Connu en anglais sous le nom de 15 Puzzle, le taquin est un jeu simple inventé dans les années 1870 aux États-Unis. Noyes Palmer Chapman, un receveur des postes de Canastota (état de New York), est probablement à l'origine du jeu, bien que le célèbre créateur de jeu Sam Loyd en ait lui aussi réclamé la paternité.

## Les règles du jeu :

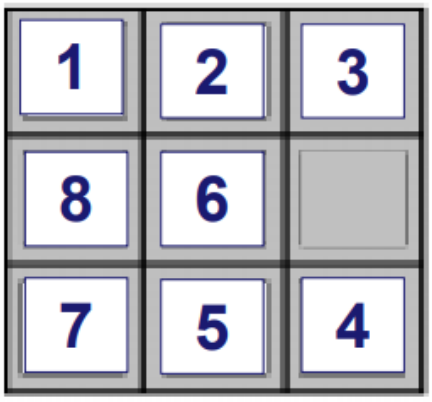
Le jeu de Taquin, est une matrice 3 × 3 (ou 4*4) contenant des cases numérotées de 1 à 8 (ou 15) et un espace vide.

Seules les cases adjacentes à l'espace vide peuvent être déplacées dans l'espace vide afin d'atteindre l'état but.

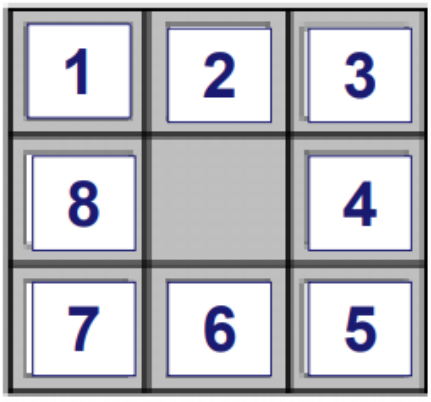
→ **L'utilisation de la recherche dans le processus de résolution de problèmes requiert une formulation abstraite du problème .**

La formulation d'un problème est définie à l'aide d'une machine à état par les différents éléments ci-après :

- **état initial :** positions des 8 plaquettes dans une des 9 cases
- **opérateurs :** déplacer la case vide
- **test-but :** état courant = état final
- **coût-chemin :** chaque déplacement coûte 1




Configuration initiale
Etat initial




Configuration finale
Etat final

# 02. Présentation des méthodes de recherche

## La recherche en largeur :

**Breadth First Search :** Consiste à parcourir le graphe par la largeur. C'est-à -dire parcourir le graphe niveau par niveau en partant de la gauche vers la droite. Dans l'exemple de la figure 1.2 ci-dessous, les chiffres en noir représentent l'ordre dans lequel les différents états seront parcouru.

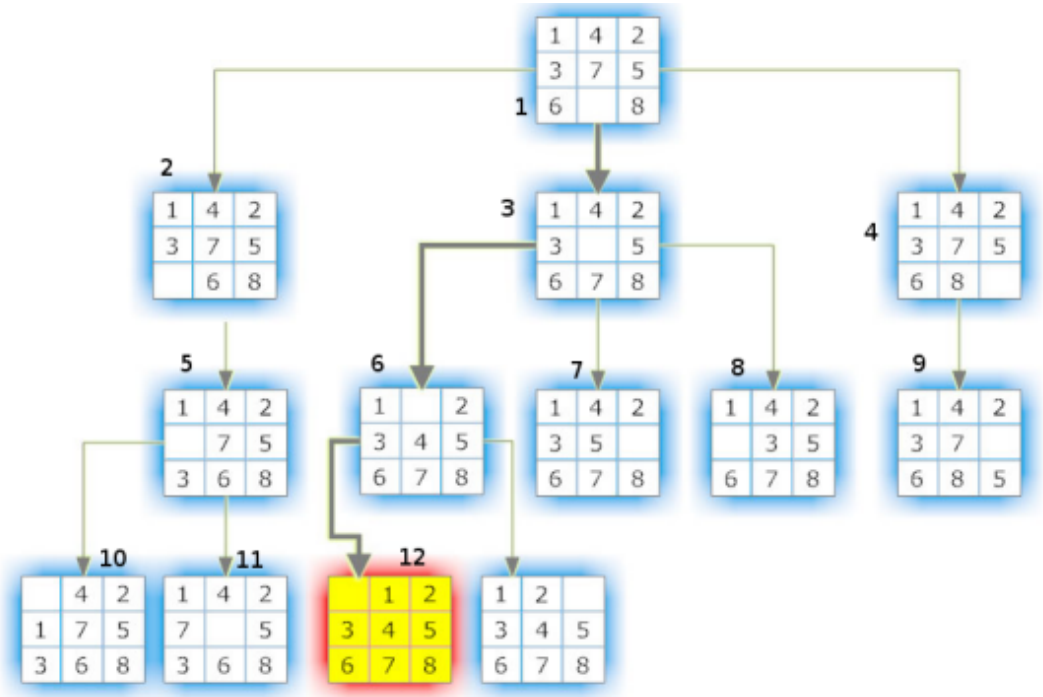
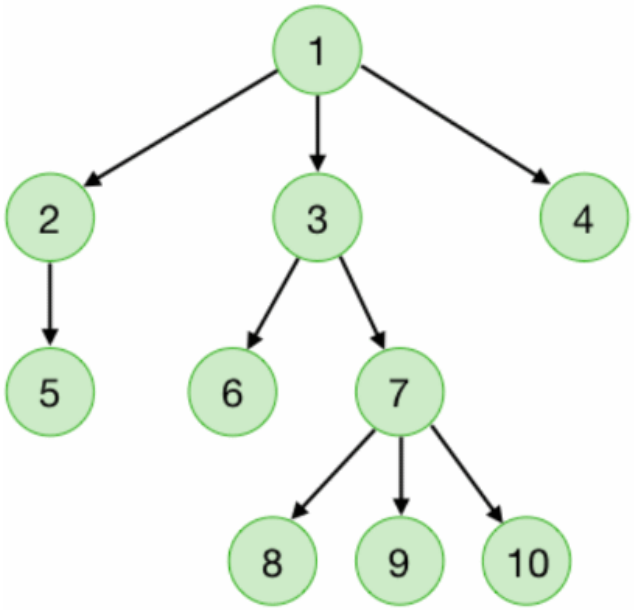



Figure 1.2 – Exemple du breadth first search pour le 8-puzzle

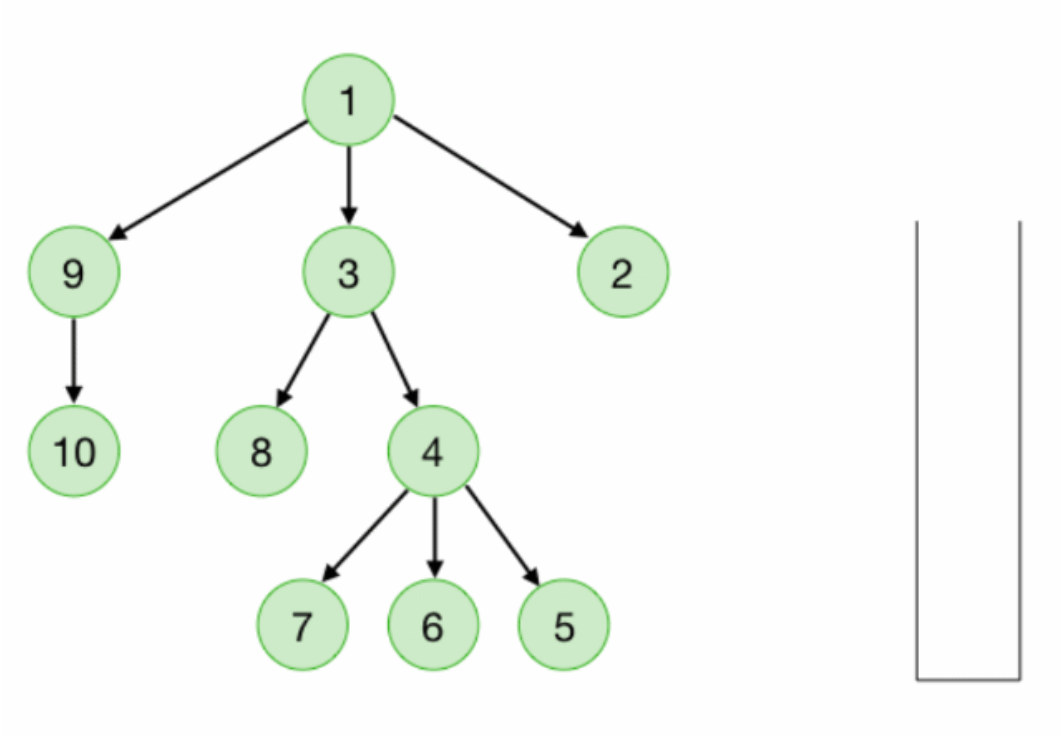
**Implémentation :** Insertion des successeurs à la fin de la file d'attente .




## La recherche en profondeur :

**Depth First Search :** 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.

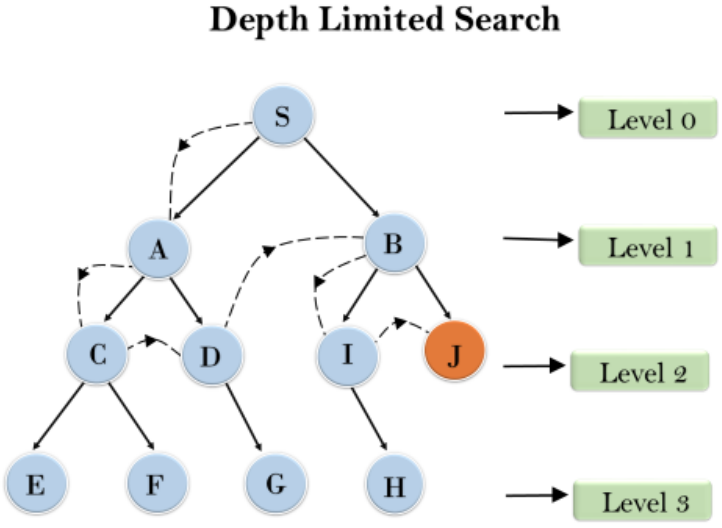
**Implémentation :** insertion des successeurs en tête de la file d'attente




## La recherche en profondeur limité :

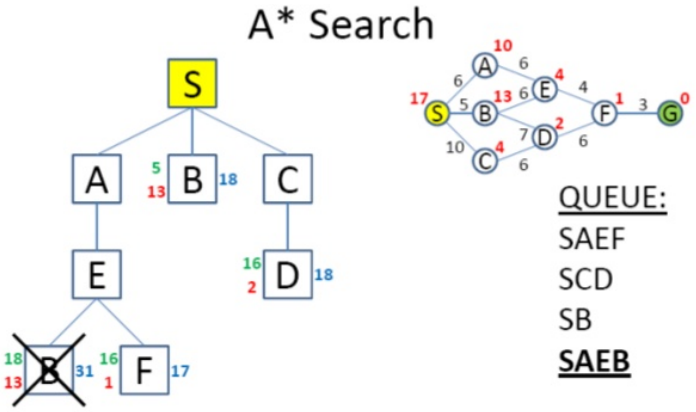
**Limited Depth First Search :** consiste à effectuer la recherche en profondeur(Depth First Search) en limitant la profondeur à atteindre lors du parcours de l'arbre de recherche.

**Implémentation :** les noeuds de profondeur L n'ont pas de successeurs



# La recherche Heuristique a* :

**A⋆ Search :** rajoute une information supplémentaire qui est la distance du nœud initial au nœud évalué, noté g(n). Il parcourt donc l'arbre l'aide d'une fonction d'évaluation h(n) pour ordonner la recherche.



# 03. implémentation de code

## Les fonctions utilisés :

| | |
|---|---|
| etat_depart | indique l'état initial du jeu . |
| estEtatFinal | retourne True si l'état passé en paramètre est l'état final. |
| position_case _vide | qui renvoie la position de la case vide dans le taquin t sous la forme d'un couple |
| permuter (t, c1, c2) | permet de permuter les pièces situées dans les cases de coordonnées c1 et c2. |

| | |
|---|---|
| afficher_taquin (t) | affiche le taquin dans l'état actuel. |
| transitions | retourne la liste des états voisins pour un état donné. Avec un coût =1 . |

## La méthode de recherche en largeur :

On utilise 2 listes principales :

- La liste des nœuds à examiner (**open**)
- La liste des nœuds déjà examinés (**closed**)

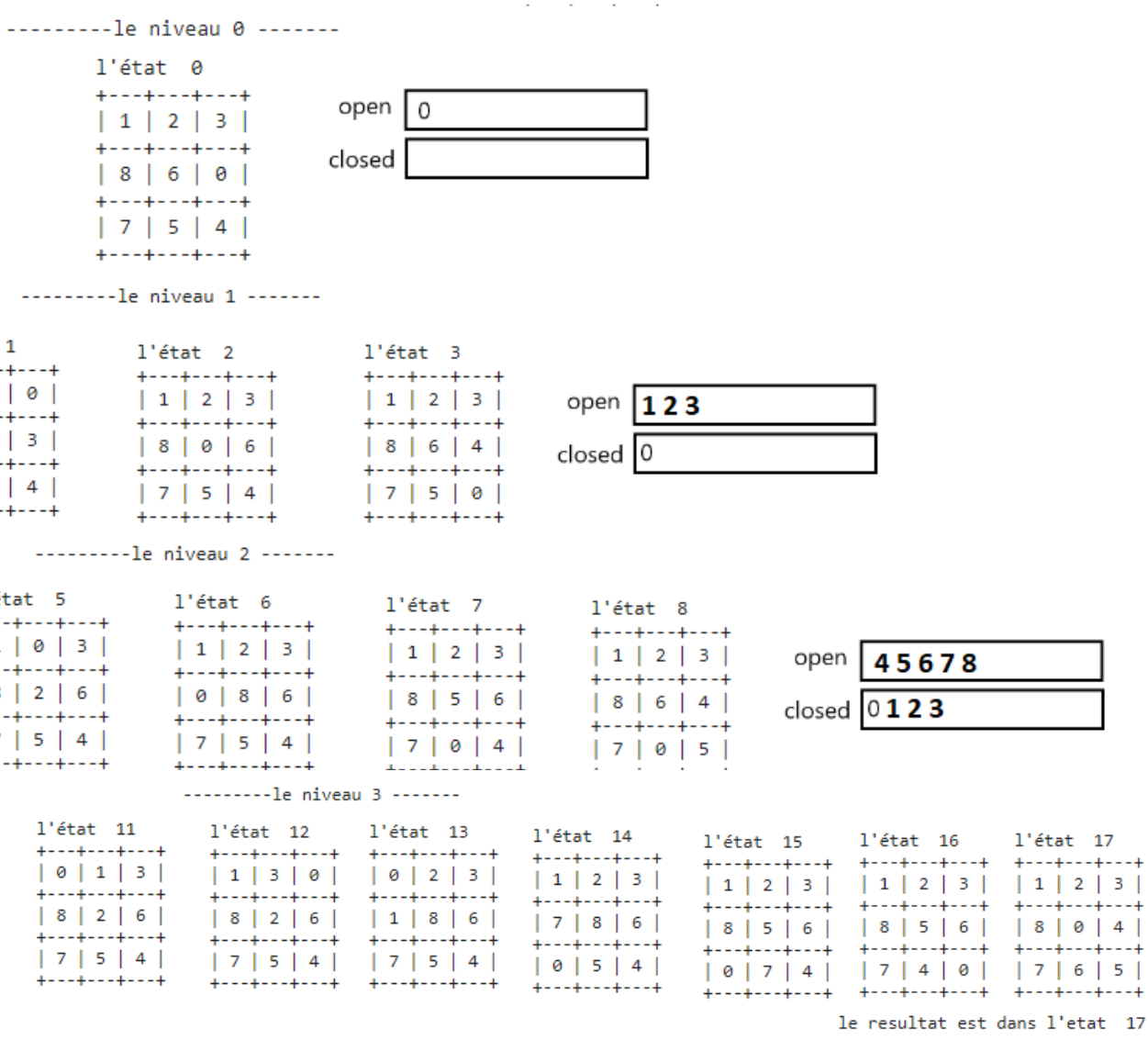
**niveau** : pour définir chaque niveau dans l'arbre
**État :** pour indiquer les numéros des matrices

⟶ A partir de la première situation t , on génère une liste dès nœuds que l'on peut
appeler childs.
Parmi ces nœuds générés, on ne garde que ceux qui ne sont pas déjà dans les listes des nœuds à traiter (open) ou des nœuds déjà traités (closed).
Les nouveaux restants (dans childs) seront ajoutés **en queue** de la liste des noeuds à traiter (open)

# Code :

```
def recherche_largeur(t, ef):
  open=[]
  closed=[]
  etat=0
  open.append(t)
  niveau=[0]
  n=0
  print ("---------le niveau",n,"-------")
  while (open != []):
    m=n
    noeud = open.pop(0)
    closed.append(noeud)
    n=niveau.pop(0)
    if n!=m :
      print ("---------le niveau",n,"-------")
    print ("l'état ",etat)
    etat+=1
    afficher_taquin(noeud)
    if (noeud==ef):
      print("le resultat est dans l'etat ",etat-1)
      return noeud
    childs = transition(noeud)
    for child in childs :
      if not child in closed :
        open.append(child)
        niveau.append(n+1)
```

# Exécution :




le traitement sera niveau par niveau :

- le niveau 0 qui contient l'état initial 0 ensuite ses fils seront dans le niveau 1

-pour l'État 1 le 0 passe en haut

-pour l'état 2 le 0 passe à droite

- pour 3 le 0 à en bas

-de même  pour les autres niveaux jusqu'à atteindre le résultat qui est dans le niveau 3  et  à l'état 17

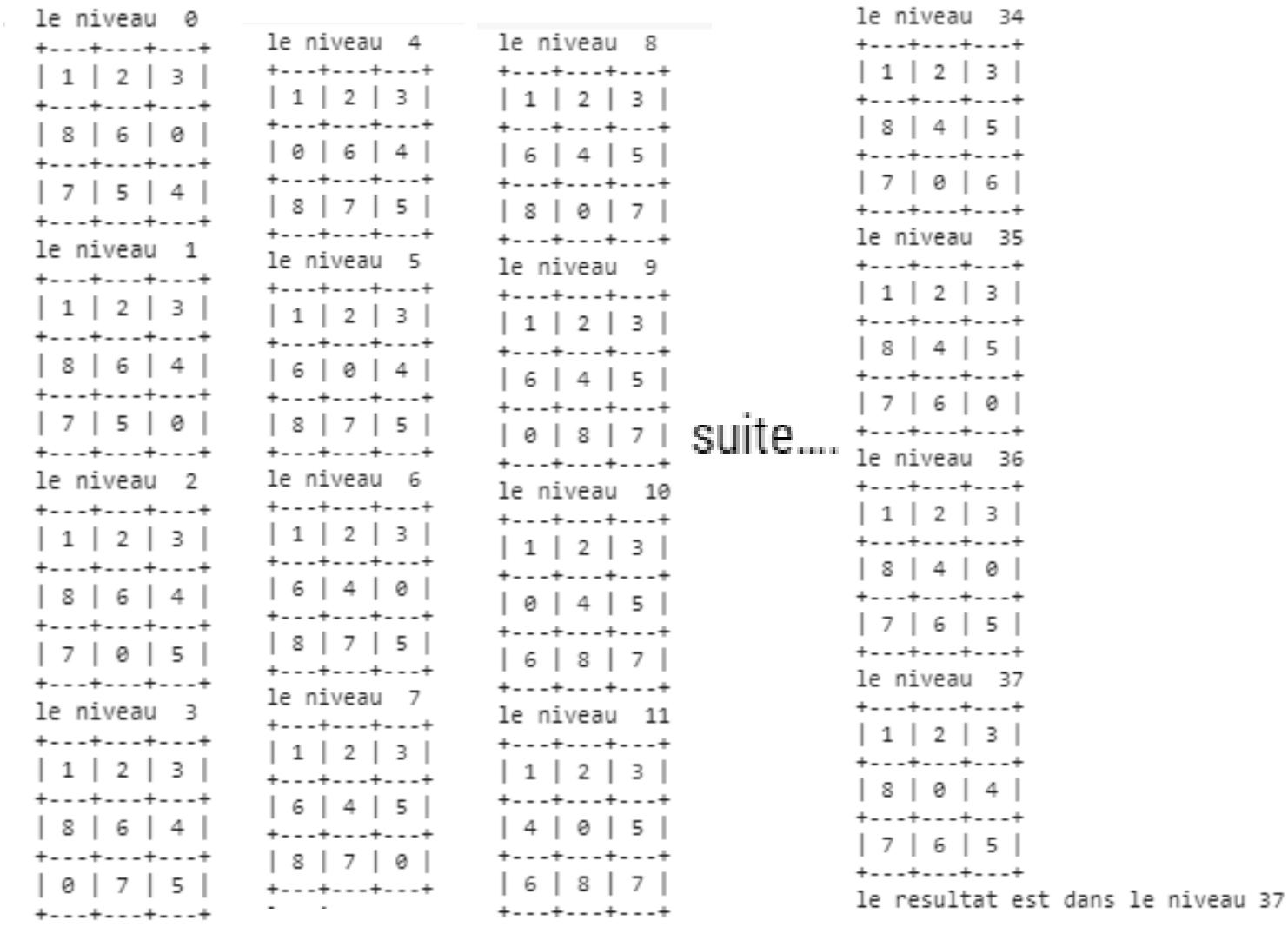
# La méthode de recherche en profondeur :

Même principe que la recherche en largeur sauf qu'on va ajouter les nouveaux nœuds générés **en tête** de la liste open.

## Code :

```
1 def recherche_profondeur(t, ef):
2   open=[]
3   closed=[]
4   niveau=0
5   open.append(t)
6   while (open != []):
7     print ("le niveau ",niveau)
8     niveau+=1
9     noeud = open.pop(0)
10    closed.append(noeud)
11    afficher_taquin(noeud)
12    if (noeud==ef):
13      print("le resultat est dans le niveau",niveau-1)
14      return noeud
15    childs = transition(noeud)
16    for child in childs :
17      if not child in closed :
18        open.insert(0,child)
19 t=[]
20 t=etat_depart()
21 ef = [[1,2,3],[8,0,4],[7,6,5]]
22 r=recherche_profondeur(t, ef)
```

# Exécution :




-le processus sera niveau par niveau sans revenir à explorer les fils :

Dans notre exemple, le résultat sera dans le niveau 37.

# La méthode de recherche en profondeur limité :

**Problème :**

En effet Le programme (profondeur d'abord) peut être perdu dans une recherche en profondeur infinie (espace d'états infini)

**Solution :**
Pour éviter ce problème on limite la profondeur de la recherche à l'aide de La méthode de recherche en profondeur limité

## Code :

On a utilisé une fonction récursive dont ces paramètres:
t= l' état initiale / open = liste des noeud visité / level = niveau dans l'arbre et le limite
On a utilisé le même principe que les autres méthodes de recherche mais en ajoutant un test de limite
et on fait un appel récursive de la fonction pour chaque nœuds générés de l'état initiale

# Exécution :

## Limite = 2 : Pas Résultat

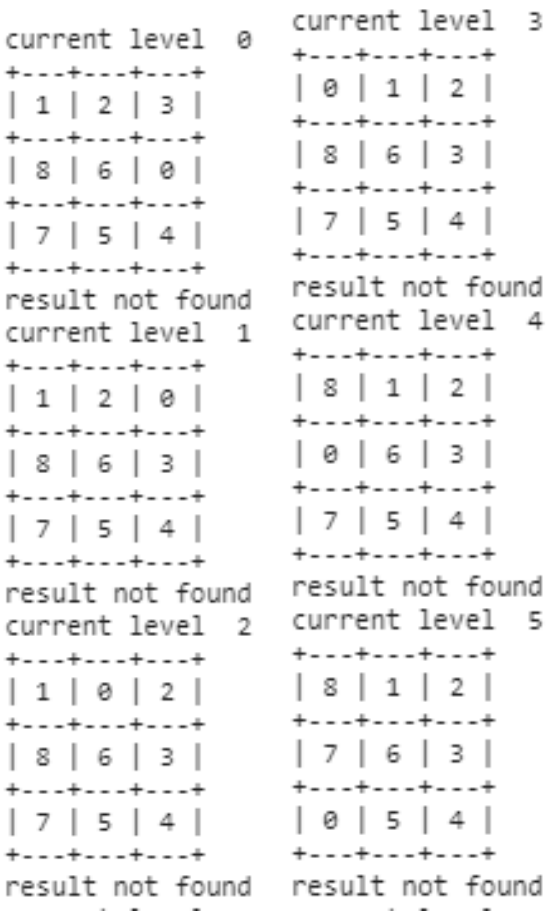
```
current level  0
+---+---+---+
| 1 | 2 | 3 |
+---+---+---+
| 8 | 6 | 0 |
+---+---+---+
| 7 | 5 | 4 |
+---+---+---+
result not found
current level  1
+---+---+---+
| 1 | 2 | 0 |
+---+---+---+
| 8 | 6 | 3 |
+---+---+---+
| 7 | 5 | 4 |
+---+---+---+
result not found
current level  2
+---+---+---+
| 1 | 0 | 2 |
+---+---+---+
| 8 | 6 | 3 |
+---+---+---+
| 7 | 5 | 4 |
+---+---+---+
result not found
```

```
current level  1
+---+---+---+
| 1 | 2 | 3 |
+---+---+---+
| 8 | 0 | 6 |
+---+---+---+
| 7 | 5 | 4 |
+---+---+---+
result not found
current level  2
+---+---+---+
| 1 | 0 | 3 |
+---+---+---+
| 8 | 2 | 6 |
+---+---+---+
| 7 | 5 | 4 |
+---+---+---+
result not found
current level  2
+---+---+---+
| 1 | 2 | 3 |
+---+---+---+
| 0 | 8 | 6 |
+---+---+---+
| 7 | 5 | 4 |
+---+---+---+
result not found
```

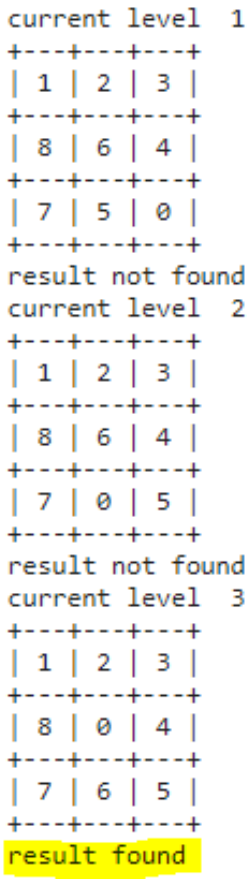
suite....

```
current level  2
+---+---+---+
| 1 | 2 | 3 |
+---+---+---+
| 8 | 5 | 6 |
+---+---+---+
| 7 | 0 | 4 |
+---+---+---+
result not found
current level  1
+---+---+---+
| 1 | 2 | 3 |
+---+---+---+
| 8 | 6 | 4 |
+---+---+---+
| 7 | 5 | 0 |
+---+---+---+
result not found
current level  2
+---+---+---+
| 1 | 2 | 3 |
+---+---+---+
| 8 | 6 | 4 |
+---+---+---+
| 7 | 0 | 5 |
+---+---+---+
result not found
False
```

# on a obtenu un résultat  pour un limite = 5




suite....




-pour l'exécution  pour  une limite égale à 2 on n'a pas trouvé un résultat alors que si on passe à un limite =5 on va trouver un résultat

# La méthode de recherche heuristique a* :

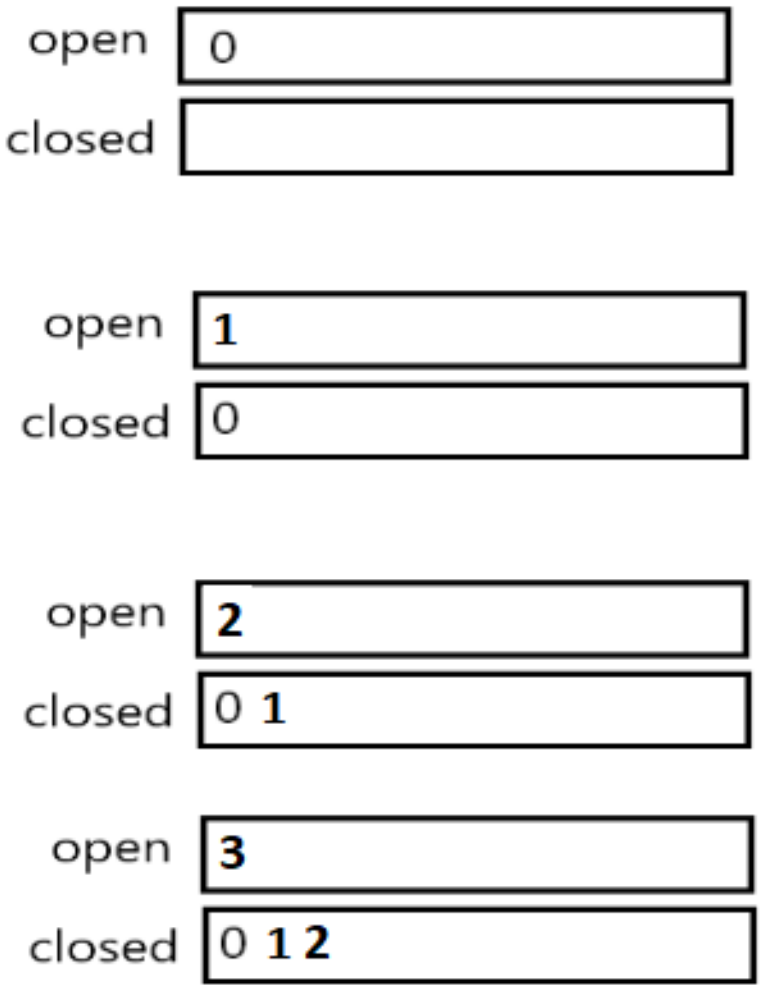
Aussi Même principe que la recherche en profondeur et en largeur sauf qu'on va ajouter une fonction d'évaluation h qui va ordonner la recherche selon le nombre des cases mal placés on utilisant un tri sur La liste des nœuds à examiner

## Code :

```
def h(t, ef):
  cpt=0
  for i in range(len(t)):
    for j in range(len(t[i])):
      if (t[i][j] != ef[i][j])and (t[i][j]!=0):
        cpt=cpt+1;
  return cpt
def recherche_heuristique (t,ef):
  open=[]
  closed=[]
  niveau=0
  success = False
  open.append(t)
  while (open != [] and not success):
    noeud = open [0]
    print ("le niveau ",niveau)
    afficher_taquin(noeud)
    niveau+=1
    open.remove(noeud)
    closed.append(noeud)
    if (noeud==ef):
      success=True
      print("le resultat est dans le niveau",niveau-1)
      afficher_taquin(noeud)
    else :
      childs = transition(noeud)
      for child in childs :
        if (child in open)or(child in closed):
          childs.remove(child)
      open= open+childs
```

# Exécution :

```
le niveau  0
+---+---+---+
| 1 | 2 | 3 |
+---+---+---+
| 8 | 6 | 0 |
+---+---+---+
| 7 | 5 | 4 |
+---+---+---+
le niveau  1
+---+---+---+
| 1 | 2 | 3 |
+---+---+---+
| 8 | 6 | 4 |
+---+---+---+
| 7 | 5 | 0 |
+---+---+---+
le niveau  2
+---+---+---+
| 1 | 2 | 3 |
+---+---+---+
| 8 | 6 | 4 |
+---+---+---+
| 7 | 0 | 5 |
+---+---+---+
le niveau  3
+---+---+---+
| 1 | 2 | 3 |
+---+---+---+
| 8 | 0 | 4 |
+---+---+---+
| 7 | 6 | 5 |
+---+---+---+
le resultat est dans le niveau 3
```




→ pour l'exécution on a trouvé une résultat juste après trois niveaux

# 04. conclusion

## La recherche en largeur : résultat trouvé

**noeuds visités :** 17 neoud

**Temps d'exécution : 4 s**

Tous les nœuds générés sont gardés en mémoire au cours de l'exécution . La mémoire est donc un véritable problème , mais trouve toujours la solution optimale

## La recherche en profondeur : résultat trouvé

**noeuds visités :** 37 noeud

**Temps d'exécution : 6 s**

Elle utilise peu de mémoire, mais peut rater la solution optimale.

## La recherche en profondeur limité :résultat trouvé

**noeuds visités :** 48 noeud

**Temps d'exécution :5 s**

Il est possible que lorsque la solution est avant les limites de la recherche.

## La recherche heuristique: résultat trouvé

**noeuds visités :** 3 noeud

**Temps d'exécution : 2s**

Elle peut trouver la solution optimale d'une manière efficace

→ **c'est la meilleur solution**

Réalisé par :
Eya Rabah
Ameni Belhadj