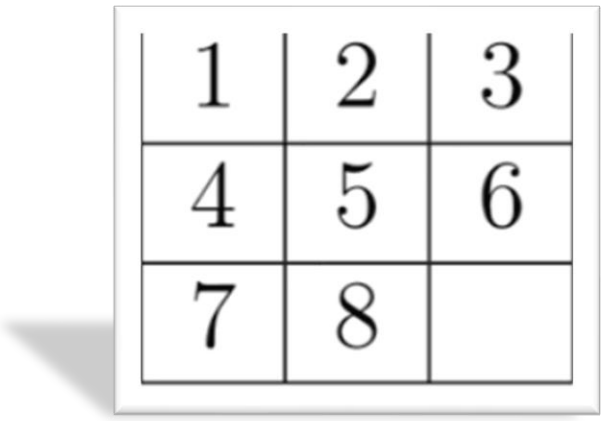


Rapport jeu taquin

- Dans ce rapport, vous trouverez différents chapitres qui expliquent la stratégie du jeu taquin, une bonne explication du fonctionnement de ce jeu et comment différents algorithmes qui seront expliqués peuvent faire une différence sur le fonctionnement du jeu, sa complexité et tout.

Rapport jeu taquin

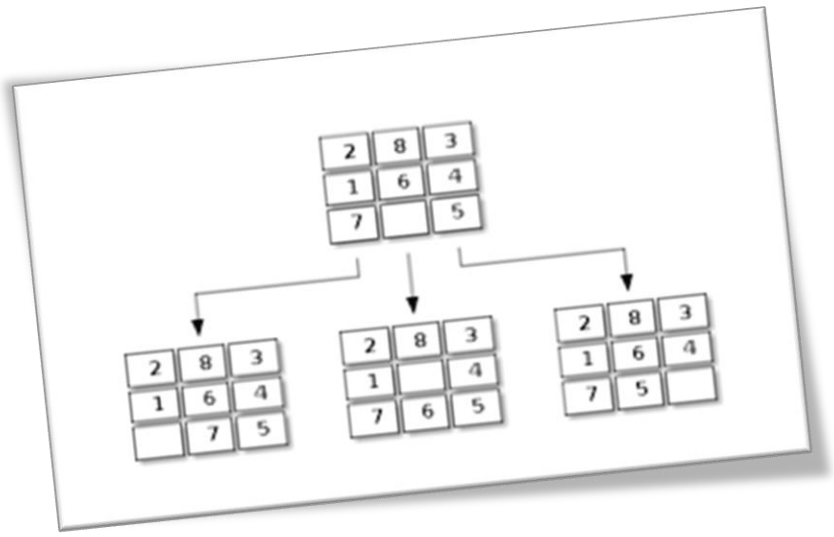
Chapitre 1 : Introduction au jeu

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

Chapitre3 : Implémentation des Algorithmes.

Chapitre4 : Comparaison entre les trois algorithmes.

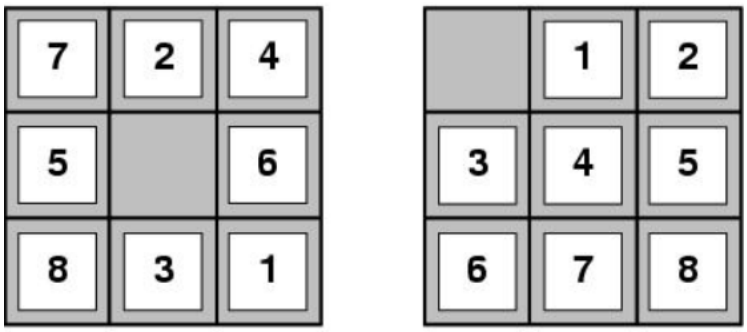
Chapitre5 : Interface graphique (Pygame)



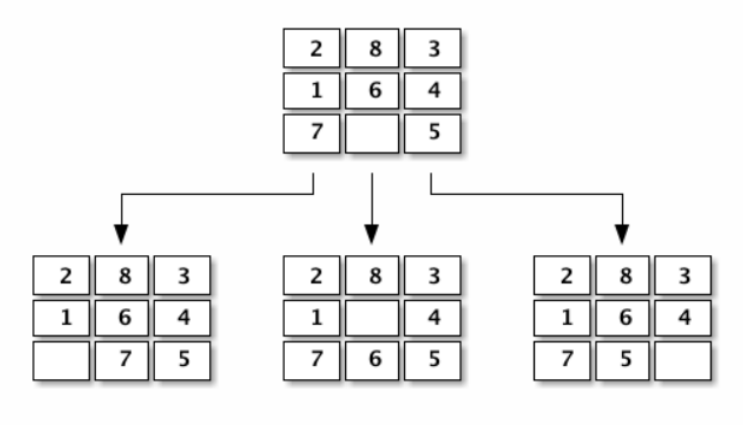
Chapitre 1 : Introduction au jeu

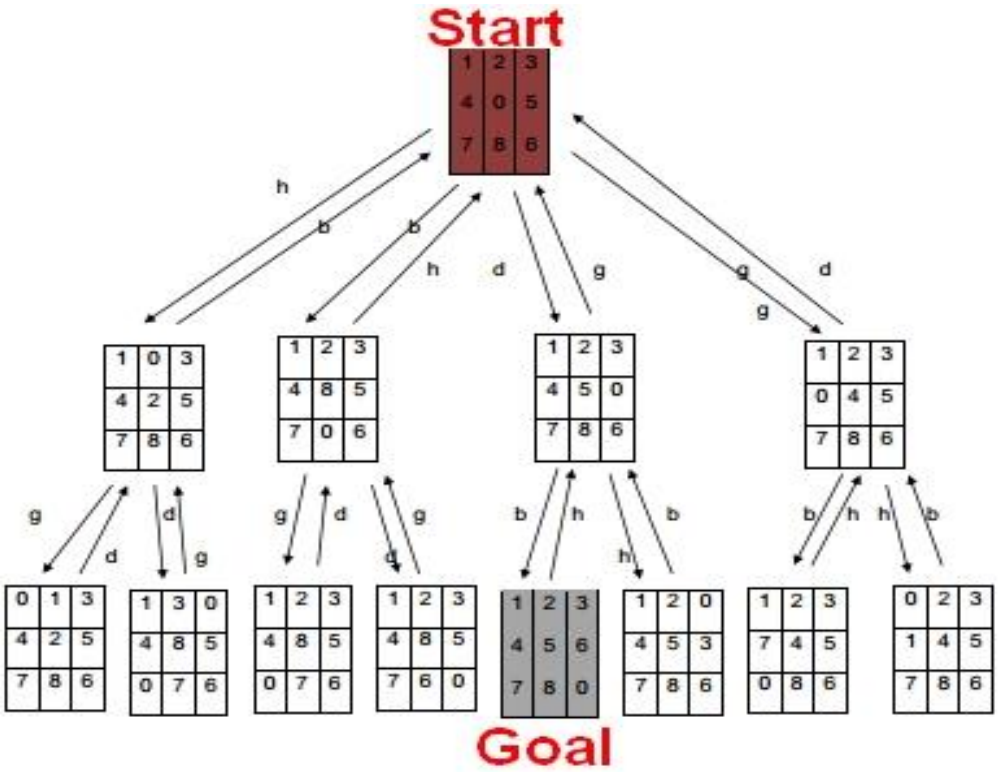
- Ce jeu, imaginé par Sam Loyd (1870), suscita immédiatement un grand intérêt à travers l'Occident. L'une des raisons peut-être de ce rapide succès était une récompense de mille dollars promis par Lloyd à quiconque parviendrait à transformer une position de départ en une position d'arrivée fixée

-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



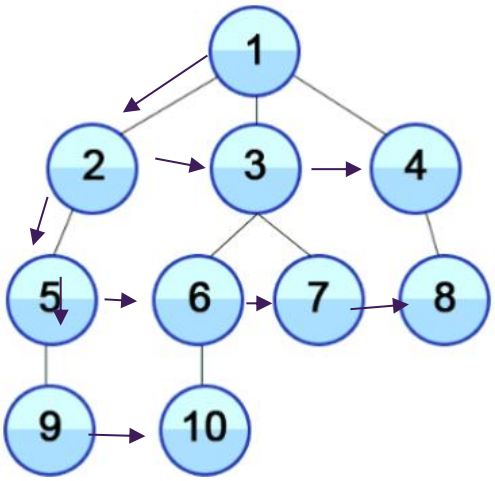


-On peut appliquer certain algorithme Pour faciliter la recherche de Solution , On va voir ça dans les chapitres suivants.

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) :

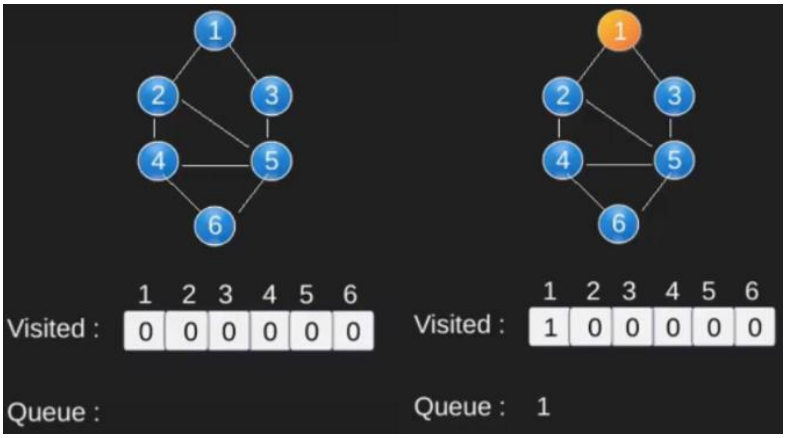


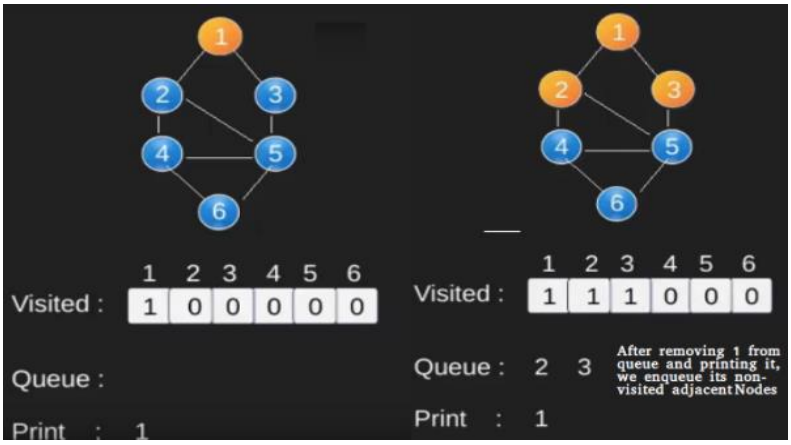
- C'est un algorithme pour parcourir ou rechercher des structures de données arborescentes ou graphiques. Il commence à la racine de l'arbre (ou à un nœud arbitraire d'un graphe) et explore tous les nœuds voisins à la profondeur actuelle avant de passer aux nœuds au niveau de profondeur suivant.

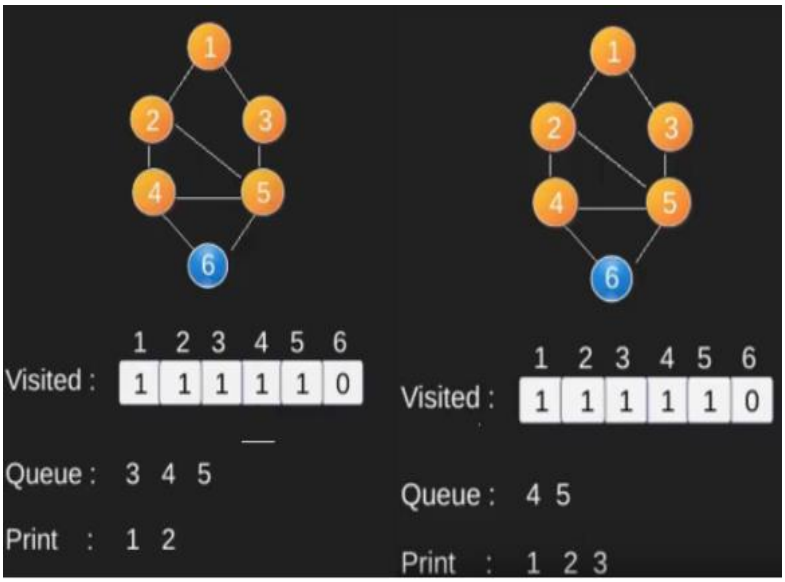
-il utilise une file d'attente (First In First Out) et il vérifie si un sommet a été découvert avant de mettre le sommet en file d'attente.

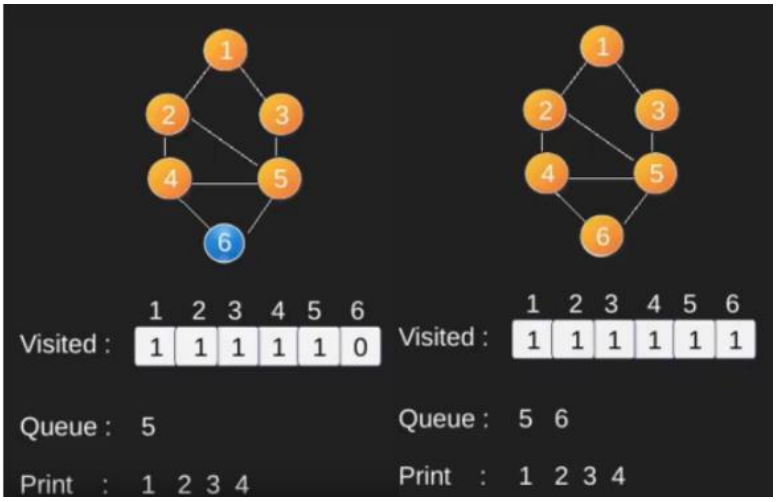
-Il utilise la stratégie opposée en tant que recherche en profondeur d'abord, qui explore d'abord les nœuds les plus profonds avant d'être forcé de revenir en arrière et d'étendre les nœuds moins profonds.

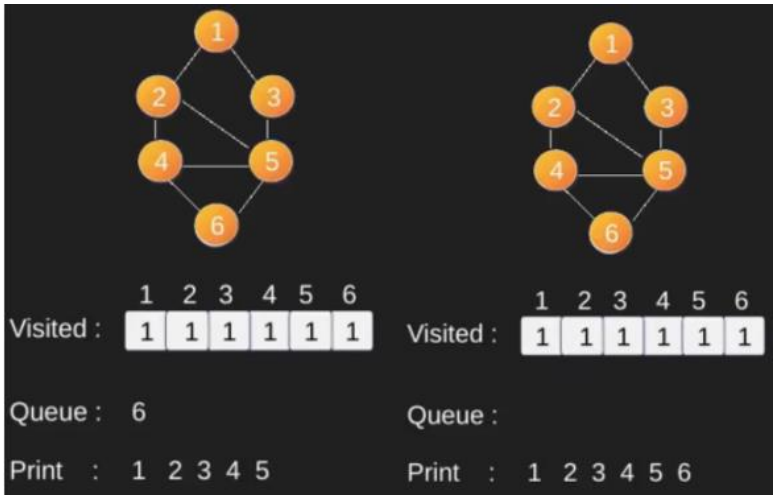
Illustrations :





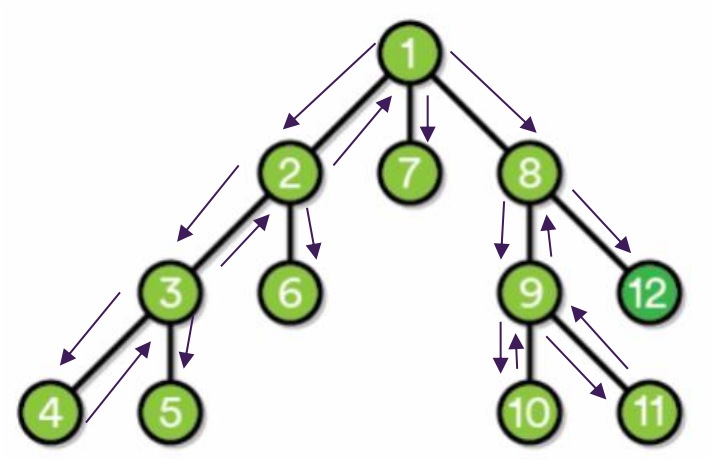






- Complexité de BFS = O (V + E) où V est les sommets et E les arêtes.

- DFS (Recherche en profondeur) :

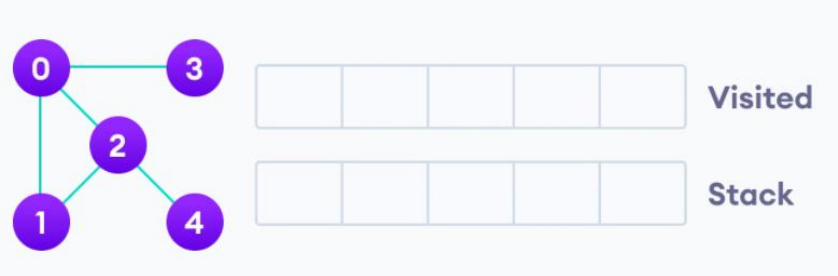


-C'est un algorithme pour parcourir ou rechercher des structures de données arborescentes ou graphiques. Il commence à la racine de l'arbre (ou à un nœud arbitraire d'un graphe) et explore tous les nœuds voisins à la profondeur actuelle avant de passer aux nœuds au niveau de profondeur suivant.

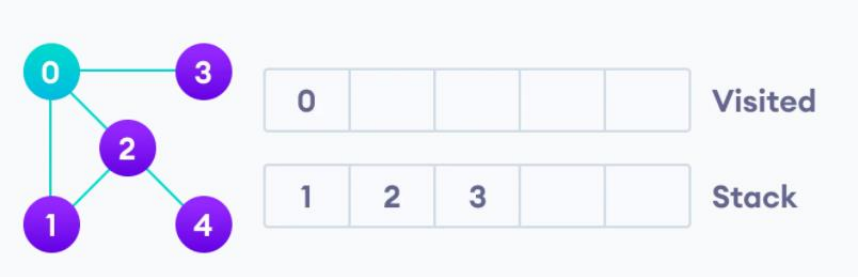
Il utilise la stratégie opposée en tant que recherche en profondeur d'abord, qui explore d'abord les nœuds les plus profonds avant d'être forcé de revenir en arrière et d'étendre les nœuds moins profonds.

L'algorithme commence au nœud racine (en sélectionnant un nœud arbitraire comme nœud racine dans le cas d'un graphe) et explore autant que possible le long de chaque branche avant de revenir en arrière

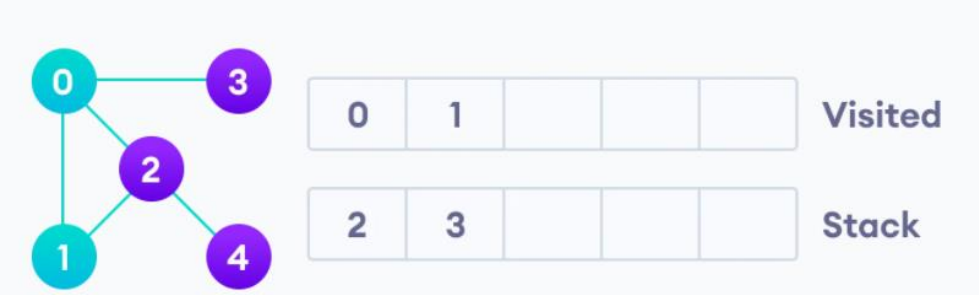
Illustrations :



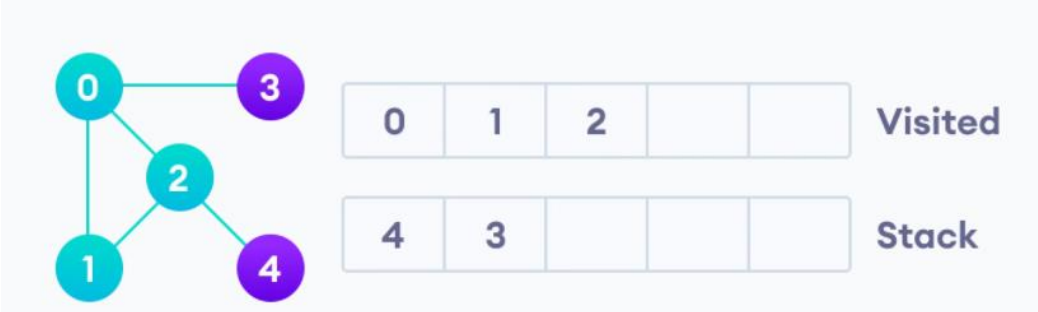
-Nous partons du sommet 0, l'algorithme DFS commence par le mettre dans la liste Visited et mettre tous ses sommets adjacents dans la pile.

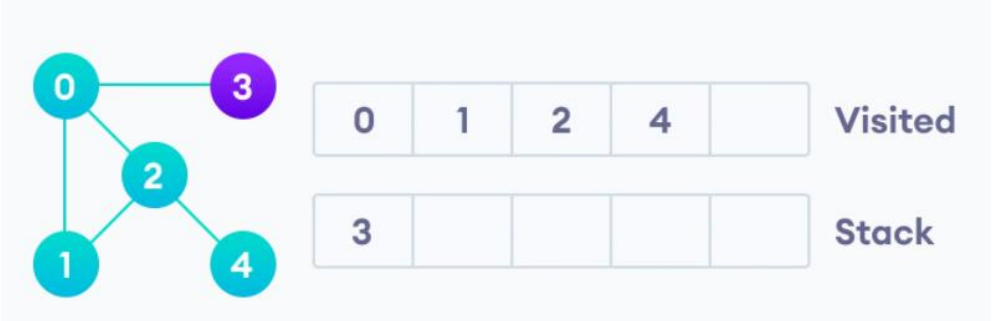


-Ensuite, nous visitons l'élément en haut de la pile, c-à-d 1 et allons à ses nœuds adjacents. Puisque 0 a déjà été visité, nous en visitons 2 à la place.



-Le sommet 2 a un sommet adjacent non visité dans 4, nous l'ajoutons donc au sommet de la pile et le visitons.





- Complexité de DFS = O(V)+O(E) = O (V + E) où V est les sommets et E les arêtes.

- A*

-C'est un algorithme de recherche * est l'une des meilleures techniques utilisées dans la recherche de chemin et les traversées de graphes.

Pourquoi un algorithme de recherche A *?

De manière informelle, les algorithmes A * Search, contrairement aux autres techniques de traversée, ont des «cerveaux». Cela signifie que c'est vraiment un algorithme intelligent qui le sépare 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).

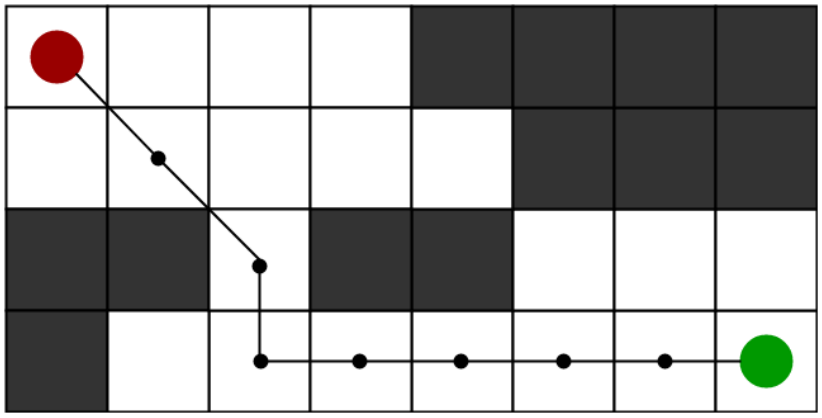


Figure 1 : Avec blocage

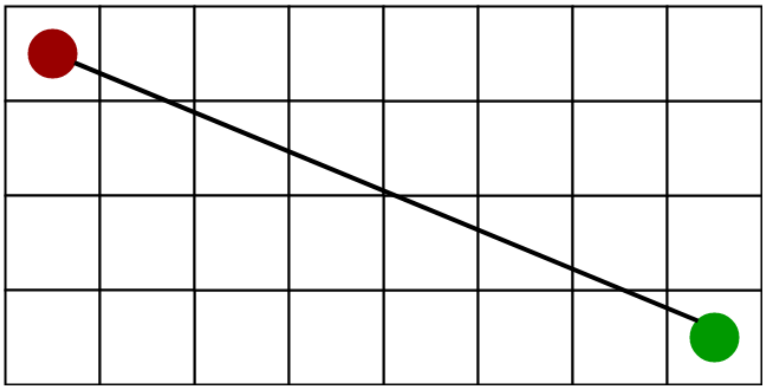


Figure 1 : Sans blocage

Chapitre3 : Implémentation des Algorithmes.

- DFS :

```
etat_initial=[[3,2,7],[8,6,0],[1,5,4]]
def estEtatFinal(t,etat_final):
  return etat_final==t
def position_case_vide (t):
  for i in range(len(t)):
    for j in range(len(t[i])):
      if t[i][j] ==0:
        return [i,j]
def numero(t, x, y):
  return t[x][y]
def afficher_taquin (t):
    for row in t:
      print('++++++++++++++')
      print('|',row[0],'|',row[1],'|',row[2],'|')
    print('++++++++++++++')
from copy import deepcopy
def permuter (t, c1, c2):
  tperm=deepcopy(t)
  a=tperm[c1[0]][c1[1]]
  tperm[c1[0]][c1[1]]=tperm[c2[0]][c2[1]]
  tperm[c2[0]][c2[1]]=a
  return tperm
def valid(x,y):
    return x>-1 and x<3 and y>-1 and y<3
def transitions(t) :
  pos=position_case_vide(t)
  tab = []
  dx = [1,-1,0,0]
  dy = [0,0,1,-1]
  for i in range(4):
    if(valid(pos[0]+dx[i],pos[1]+dy[i])):
       tab.append([pos[0]+dx[i],pos[1]+dy[i]])
```

```
    nvmatrice=[]
    for i in tab :
      nvmatrice.append(permuter(t,pos,i))
    return nvmatrice
trace= []
visited= []
success= False
etat=[[3,2,7],[8,6,4],[1,0,5]]
def dfs(node, etat):
  global success
  if (success==False and node not in visited):
    if estEtatFinal(node,etat):
      trace.append(node)
      success=True
    trace.append(node)
    visited.append(node)
    tab=transitions(node)
    for w in tab:
      if w not in visited and success==False:
         dfs(w,etat)
dfs(etat_initial,etat)
for i in trace :
  afficher_taquin (i)
```

Explication d'algorithme DFS :

- Création d'un tableau visited, un tableau path, une variable boolean success initialiser à false.
- Générer dans chaque fonction les états fils(les transitions).
- Parcourir tous les fils d'un nœud (qui ne sont pas encore visités) en appelant à chaque fois notre fonction qui va recevoir comme paramètre le nouvel état fils et l'état final qu'on cherche et en mettant visited[nœud]=true.
- Si on arrive à l'état final qu'on cherche on met success sur true et on ne fait plus d'appels.

- BFS :

```
etat_initial=[[3,2,7],[8,6,0],[1,5,4]]
def estEtatFinal(t,etat_final):
  return etat_final==t
def position_case_vide (t):
  for i in range(len(t)):
    for j in range(len(t[i])):
      if t[i][j] ==0:
        return [i,j]
def numero(t, x, y):
  return t[x][y]
def afficher_taquin (t):
    for row in t:
      print('+++++++++++++')
      print('|',row[0],'|',row[1],'|',row[2],'|')
    print('+++++++++++++')
from copy import deepcopy
def permuter (t, c1, c2):
  tperm=deepcopy(t)
  a=tperm[c1[0]][c1[1]]
  tperm[c1[0]][c1[1]]=tperm[c2[0]][c2[1]]
  tperm[c2[0]][c2[1]]=a
  return tperm
def valid(x,y):
    return x>-1 and x<3 and y>-1 and y<3
def transitions(t) :
  pos=position_case_vide(t)
  tab = []
  dx = [1,0,-1,0]
  dy = [0,1,0,-1]
  for i in range(4):
    if(valid(pos[0]+dx[i],pos[1]+dy[i])):
       tab.append([pos[0]+dx[i],pos[1]+dy[i]])
```

```
30.     for i in range(4):
31.       if(valid(pos[0]+dx[i],pos[1]+dy[i])):
32.          tab.append([pos[0]+dx[i],pos[1]+dy[i]])
33.     nvmatrice=[]
34.     for i in tab :
35.       nvmatrice.append(permuter(t,pos,i))
36.     return nvmatrice
37.   trace= []
38.   visited= []
39.   success= False
40.   etat=[[3,2,7],[8,6,4],[1,0,5]]
41.   def bfs(node):
42.       global success
43.       global etat
44.       global visited
45.       global trace
46.       q = []
47.       q.append(node)
48.       while(len(q) != 0 and success==False):
49.           p=q.pop(0)
50.           trace.append(p)
51.           visited.append(p)
52.           if estEtatFinal(p,etat):
53.               success=True
54.           else:
55.               tab=transitions(p)
56.               for w in tab:
57.                   if (w not in visited and success==False):
58.                       q.append(w)
59.   bfs(etat_initial)
60.   for i in trace :
```

Explication d'algorithme BFS :

- La Création d'un tableau visited, d'un tableau path, une variable boolean success initialiser à false.
- La Création d'une liste qu'on va utiliser dans fonction bfs comme étant une file.
- Enfiler notre premier état dans la file et puis tant on est pas arrivé à l'état final et que la file n'est pas vide, on défile la file et en prend notre état courant pour générer dans chaque fonction les états fils(les transitions).
- Si on arrive à l'état final qu'on cherche on met success sur true et on ne fait plus l'appels.
- Parcourir tous les fils d'un nœud (qui ne sont pas encore visités) en enfilant à chaque fois notre état dans la file et en mettant visited[nœud]=true.

A* :

```
etat_initial=[[3,2,7],[8,6,0],[1,5,4]]
def estEtatFinal(t,etat_final):
  return etat_final==t
def position_case_vide (t):
  for i in range(len(t)):
    for j in range(len(t[i])):
      if t[i][j] ==0:
        return [i,j]
def numero(t, x, y):
  return t[x][y]
def afficher_taquin (t):
    for row in t:
      print('+++++++++++++')
      print('|',row[0],'|',row[1],'|',row[2],'|')
    print('+++++++++++++')
from copy import deepcopy
def permuter (t, c1, c2):
  tperm=deepcopy(t)
  a=tperm[c1[0]][c1[1]]
  tperm[c1[0]][c1[1]]=tperm[c2[0]][c2[1]]
  tperm[c2[0]][c2[1]]=a
  return tperm
def valid(x,y):
    return x>-1 and x<3 and y>-1 and y<3
def transitions(t) :
  pos=position_case_vide(t)
  tab = []
  dx = [1,-1,0,0]
  dy = [0,0,1,-1]
  for i in range(4):
    if(valid(pos[0]+dx[i],pos[1]+dy[i])):
       tab.append([pos[0]+dx[i],pos[1]+dy[i]])
  nvmatrice=[]
```

```
34.   for i in tab :
35.     nvmatrice.append(permuter(t,pos,i))
36.   return nvmatrice
37. etat_final=[[3,2,7],[8,6,4],[1,0,5]]
38. 
39. def aStar(start):
40.     #The open and closed sets
41.     openset = set()
42.     closedset = set()
43.     G={}
44.     H={}
45.     parent={}
46.     current = start
47.     CurrentToStr = ''
48.     for i in range(3):
49.         for j in range(3):
50.             CurrentToStr=CurrentToStr+str(current[i][j])
51.     openset.add(CurrentToStr)
52.     parent[CurrentToStr]=CurrentToStr
53.     #While the open set is not empty
54.     while openset:
55.         current = min(openset, key=lambda o:D[o] + H[o])
56.         if current == etat_final:
57.             path = []
58.             while parent[CurrentToStr]!=CurrentToStr:
59.                 path.append(CurrentToStr)
60.                 CurrentToStr= parent[CurrentToStr]
61.             path.append(CurrentToStr)
62.             return path[::-1]
63.         openset.remove(CurrentToStr)
64.         closedset.add(CurrentToStr)
65.         for node in transitions(current):
66.             NodeToStr=""
```

```
67.                 for i in range(3):
68.                     for j in range(3):
69.                         NodeToStr=NodeToStr+str(node[i][j])
70.                 if NodeToStr in closedset:
71.                     continue
72.
73.                 if NodeToStr in openset:
74.                     new_g = G[CurrentToStr] + 1
75.                     if G[[NodeToStr] > new_g:
76.                         G[NodeToStr] = new_g
77.                         parent[NodeToStr] = CurrentToStr
78.                 else:
79.                     G[NodeToStr] = G[CurrentToStr] + 1
80.                     H[NodeToStr]=0
81.                     for i in range(3):
82.                         for j in range(3):
83.                             if(node[i][j]!=etat[i][j]):
84.                                 H[NodeoStr]=H[NodeoStr]+1
85.                     parent[NodeToStr] = CurrentToStr
86.                     openset.add(NodeToStr)
87. print(aStar(etat_initial))
```

Explication d'algorithme A* :

- Notre approximation (probabilité) va être basé sur le nombre de cases mal placés.
- La Création d'un dictionnaire parent, une set closed et une set open: (la set est plus efficace que la liste quand on cherche un élément au sein de la structure de donnée):
- Créer deux dictionnaires H (heuristique) et G.
- Transformer notre état qui est sous forme de tableau 3*3 en une chaine de caractère car les dictionnaires n'acceptent pas les listes 3*3 comme clé.
- Affecter G[état initial]=0
- Compter le nombre des états mal placés et affecter à H[état initial]=nombre d'états mal placés.
- Tant que notre open set n'est pas vide on va chercher l'état qui le minimi G+H et lui affecter à notre état courant, si on arrivera l'état final on écrit notre path à l'aide du dictionnaire parent parent qui va être retourner par notre fonction si non on enlève notre état courant de la open set et on lui ajoute A closed set, on génère par la suite nos transitions et on les parcoure, si le nouveau état est inclus déjà dans la closed set on dépasse cet état puis on vérifie si il appartient à notre open set et on cherche si cet état peut être achever par notre état courant avec une approximation mois que la précédente si c'est le cas on le met à jour si non on le dépasse et puis si c'est le cas ou notre état n'appartient ni à la closed set ni à la open set on lui règle ces données(parent,approximation) et on l ajoute a la open set.

Chapitre4 : Comparaison entre les trois algorithmes.

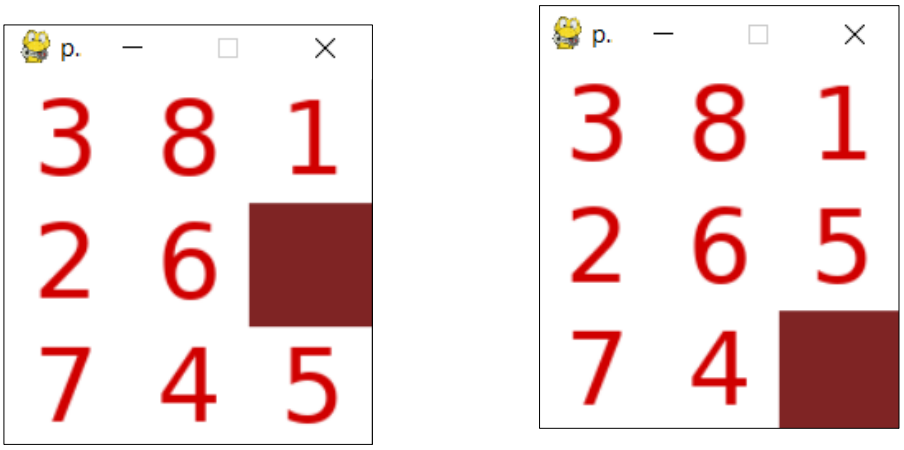
	DFS	BFS	A*
DFS		Si l'arbre a un grand facteur de ramification (BFS devient lent avec des arbres larges) Si des solutions et fréquentes et situées au fond de l'arbre S'il y a une limite à la quantité de mémoire pouvant être utilisée	Si l'heuristique de A * est mauvaise Si le but est l'optimalité et que l'heuristique A * n'est pas admissible Si les solutions sont fréquentes et situées au plus profond de l'arbre
BFS	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 Si vous savez que la solution est maintenant loin de la racine de l'arbre Si les solutions sont rares et situées au		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

	plus profond de l'arbre Quand l'arbre est clairsemé (je ne sais pas pourquoi)		
A*	Si l'arbre est infini Si l'arbre est dense En général, la recherche aveugle est plus lente que la recherche heuristique, donc pour une heuristique assez bonne, A * doit être préféré	Si l'espace mémoire est limité Si l'arbre a un facteur de ramification élevé Si l'arbre est dense 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 généralement meilleure que la complexité temporelle de BFS avec une heuristique assez bonne	

Comparaison entre les trois algorithmes (Autres stratégies) :

Algorithme	Complet?	Optimal?	Complexité temps	Complexité espace
DFS	Non	Non	O(b^m)	O(bm)
BFS	Oui	Oui si cout =1	O(b^d)	O(b^d)
A*	Oui	Oui si l'heuristique est inférieur ou égal à la vérité	Nombre des nœuds avec g(n)+h(n)<=C*	

Chapitre5 : Interface graphique (Pygame)



1. D'abord il faut importer les bibliothèques.

```
import pygame,sys,random
from pygame.locals import *
```

2. pygame. Init () initialise tous les modules pygame importés et on applique la taille de la fenêtre.

```
pygame.init()
fen=pygame.display.set_mode((181,179))
```

3. Chargement des images (pour L'affichage graphique

```
def initImages():
    images=[]
    for i in range(9):
        images.append(pygame.image.load("t"+str(i)+".png"))
    return images
```

-Les images sont dans le dossier où se trouve les fichiers. Py

4.On modifie la fonction affichage pour qu'elle affiche les images (fen.blit)

fenetre.blit(image1,image1_rect) on blitte l'image dans ce rectangle.

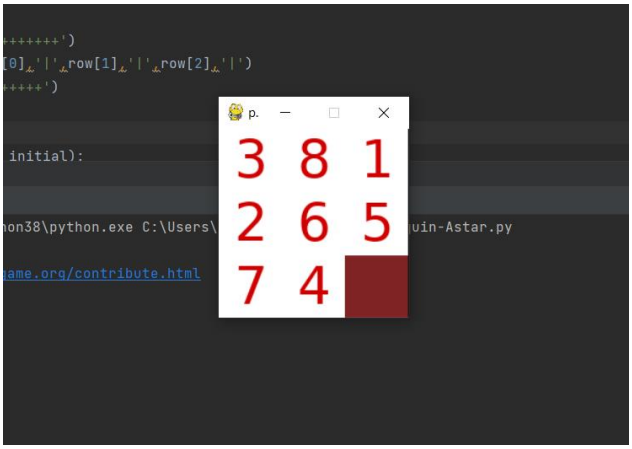
```
def affiche(tableau,images):
    for i in range(3):
        for j in range(3):
            numeroCase=tableau[i][j]
            case=images[numeroCase-1]
            fen.blit(case,(i*60,j*60))
```

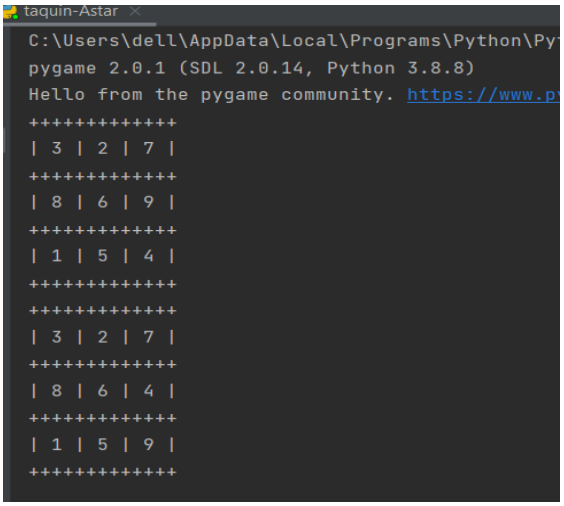
5.on prépare les algorithme que se soit A*, DFS, BFS.

```
dfs(etat_initial,etat_final)
tableau = etat_initial = [[3, 2, 7]
for k in trace:
    nb=nb+1
    affiche(k,images)
    afficher_taquin(tableau)
        #for evenement in pygame.ev
        #    affiche(tableau, images
        #    if evenement.type == QU
            #    pygame.quit()
                #sys.exit()
    pygame.display.update()
    pygame.time.wait(2000)
    if(nb==len(trace)):
        pygame.time.wait(20000)
```

```
for k in aStar(etat_initial):
    nb=nb+1
    tableau= string_to_array(k)
    affiche(tableau,images)
    afficher_taquin(tableau)
        #for evenement in pygame.event.get():
        #    affiche(tableau, images)
        #    if evenement.type == QUIT:
            #    pygame.quit()
                #sys.exit()
    pygame.display.update()
    pygame.time.wait(2000)
    if(nb==len(aStar(etat_initial))):
        pygame.time.wait(20000)
```

6.Affichage sur le terminal et graphique en parallèle (2 fonctions d'affichage)





7. pygame.display.update () pour que la surface d'affichage apparaisse réellement sur le moniteur de l'utilisateur.

```
pygame.display.update()
```

8.Pour prolonger la durée de l'affichage on applique

La fonction Pygame.time.wait(1second*1000)

```
pygame.time.wait(2000)
if(nb==len(trace)):
    pygame.time.wait(20000)
```

Annex :

https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/

https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/

https://www.geeksforgeeks.org/a-search-algorithm/

https://fr.wikipedia.org/wiki/Taquin