

Mini projet Intelligence Artificielle

Rapport du problème de Taquin

Réalisé par :
YARMANI Yosra - LZCS01
et
CHELLY Safa - LZCS03

1	2	7
8	5	
3	5	4

Initialisation

	Profondeur	Largeur	Profondeur limité	A*
Nœuds visités				

Année Universitaire :

2021 - 2022

Institut Supérieur d'Informatique - ISI Ariana

Plan

Chapitre 1 : Introduction au problème

Chapitre 2 : Les algorithmes de résolution:

Recherche en profondeur

Recherche en largeur

Recherche A*

Recherche en profondeur limitée

Chapitre 3 : Implémentation des algorithmes

Chapitre 4 : conclusion.

Introduction

1	2	3
4	5	6
7	8	

Le jeu de taquin se compose d'un cadre contenant neuf cases carrées numérotées de 1 à 9,

laissant un espace dans lequel on peut faire coulisser une plaque voisine.

Ce jeu consiste à remettre dans l'ordre les cases du taquin à partir d'une configuration initiale.

Problème proposé :

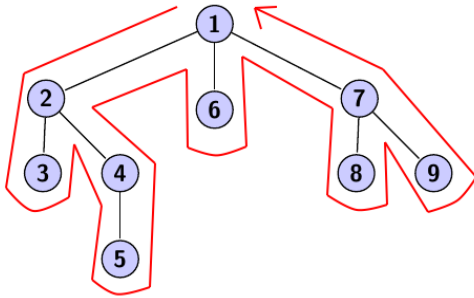
On dispose d'un taquin de taille 3×3 , dont les cases sont numérotées de 1 à 8.

Dans ce projet, on va réaliser un programme écrit en Python pour la résolution du jeu du taquin en utilisant les algorithmes de recherche qui sont déjà vus en cours.

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

Les Algorithmes de Résolution (1)

Recherche en profondeur (Depth-First-Search; DFS) :



L'algorithme de parcours en profondeur est un algorithme de parcours d'arbre, et plus généralement le parcours de graphe. Son application consiste à déterminer s'il existe un chemin d'un sommet à un autre.

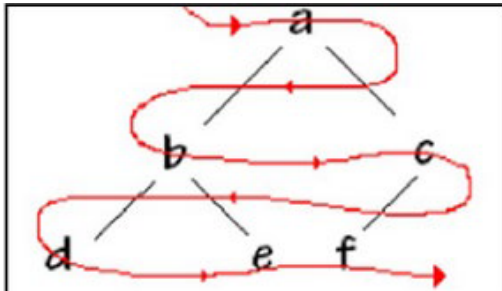
L'exploration d'un parcours en profondeur depuis un sommet **s** fonctionne comme suit:

Il poursuit un chemin dans le graphe jusqu'à atteindre un sommet déjà visité. Il revient alors sur le dernier sommet où on pouvait suivre un autre chemin puis explorer un autre chemin. L'exploration s'arrête quand tous les sommets depuis '**s**' ont été visités.

→ l'exploration progresse à partir d'un sommet **s** en s'appelant récursivement pour chaque sommet voisin de **s**.

Les Algorithmes de Résolution (2)

Recherche en Largeur (Breadth First Search; BFS) :



L'algorithme de parcours en largeur permet le parcours d'un graphe ou d'un arbre de la manière suivante:

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 :


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

→ **Ce mode de fonctionnement utilise donc un file dans laquelle il prend le premier sommet et place en dernier ses voisins non encore explorés. Les nœuds déjà explorés sont marqués afin d'éviter qu'un même nœud soit exploré plusieurs fois.**

Implémentation des algorithmes (1)

- **Les Fonctions Basiques:**

- La case contenant le numéro **0** représente 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.
1. Importation de la fonction **"deepcopy"** de la bibliothèque **"copy"** qui copie de manière récursive un objet dans un autre objet; c'est à dire construire un nouvel objet composé puis, récursivement, insérer des copies des objets trouvés dans l'objet original
 2. Importation de la bibliothèque **"time"**; qui permet de récupérer un temps.

```
✓ 0s  #importation de la bibliotheque dont on a besoin dans notre programme
from copy import deepcopy
import time
```

3. **Nos variable globales "t" et "tf"**

- **Configuration initiale:** **t** = [[1,2,3],[8,6,0],[7,5,4]]
- **Configuration finale:** **tf** = [[1,2,3],[8,0,4],[7,6,5]]

```
t = [[1,2,3],[8,6,0],[7,5,4]] #C'est une liste qui représente l'état de départ ou initial
tf = [[1,2,3],[8,0,4],[7,6,5]] #est une liste qui représente l'état finale ou but
```

4. **Fonction EstEtatFinal :**

Cette fonction retourne une liste qui représente notre état initial.

```
#initialisation d'une fonction dont son retour est une variable boolenne qui nous renseigne sur l'acheminement de l'etat final
def estEtatFinal (t,tf):
    return t==tf
```

Implémentation des algorithmes (2)

5. Fonction position_case_vide :

```
#initialisation d'une fonction dont son retour est une variable qui nous renseigne sur la position de la case vide
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)
```

6. Fonction Permuter :

```
#initialisation d'une fonction qui permute les cases et retour le nouveau taquin
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] # permutation, the python way
    return temp
```


Implémentation des algorithmes (3)

- Implémentation de la fonction de recherche en largeur:

```
[ ] def bfs(t, tf):
    global start
    start=time.time()#      Start: récupère le temps de lancement de l'exécution
    visited=[] #            Visited: une liste qui contiendra les états visités
    queue=[] #             Queue: une liste qui contiendra les états à parcourir

    queue.append(t) #       On ajoute l'état initial à queue
    visited.append(t) #     On ajoute l'état initial à visited
    trace=[] #              Trace une liste qui contiendra les états parcourus partant de l'état initial à l'état final (Elles sont initialisées à vide)

    while queue: #          queue n'est pas vide
        f=queue.pop(0) #    f reçoit le 1er élément de queue et l'élimine de cette liste

        if estEtatFinal(f,tf):# On teste si f coïncide avec l'état final ou non.
            trace.append(f)#   on ajoute f à trace
            success=True#      et success devient true
            break
        trace.append(f) #    Sinon on ajoute f à trace
        trs=transitions(f) # trs reçoit les transitions possibles de f
        for j in trs: #       On parcourt ces transitions l'une après l'autre jusqu'à queue devient vide
            if j not in visited: # s'elles ne sont pas dans visited
                visited.append(j)# on les ajoute dans queue et dans visited
                queue.append(j)
    return trace

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

- Implémentation de la fonction de recherche en profondeur:

```
[ ] trace= [] #            Initialisation de la liste des traces
#
# Visited= [] #            Trace contient les états parcourus par la traque de l'état initial à l'état final
# success= False #        Initialisation de la liste des noeuds visités
#                          Success initialisée à False donne True si l'état final est atteint

def dfs(t, tf):
    global start
    start= time.time()
    global success
    if (success==False and t not in visited):
        # si l'état actuel n'est pas encore visité et on n'a pas atteint l'état but on ajoute cet état à trace
        trace.append(t)
        if estEtatFinal(t,tf):#si c'est l'état final, success devient True
            success=True
        visited.append(t) #   On ajoute l'état actuel à visited
        tab=transitions(t) #  la variable tab contient les transitions possibles à partir de l'état actuel
        for i in tab:
            #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 dès 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
            if i not in visited and success==False:
                dfs(i,tf)
        #appel a la fonction de resolution par methode de profondeur
        dfs(t,tf) #on appelle après la fonction dfs pour s'exécuter et on affiche tous les états parcourus dans trace et leur niveau.
    niveaux=0
    for i in trace :
        print()
        print ('** pas', niveaux, ' : **')
        afficher(i)
        niveaux+=1
    print("\n --> Goal trouvé après",niveaux-1," iterations .")
    print ('Time spent: %0.2fs' % (time.time()-start))
```

Implémentation des algorithmes (4)

- Implémentation de la fonction de recherche A*:

```
def h(t,tf):
    """
    la fonction heuristique « h » qui retourne le nombre des cases mal placées sans tenir compte du case vide.
    """
    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

def aetoile(t,tf):
    success=False# success initialisée à false indique si on a atteint l'état but ou non.
    start = time.time() # start repère le début de temps d'exécution
    niveaux = 0 # Niveaux indique le niveau de cet état (le nombre de transitions faites à partir de l'état initial
    free_nodes = [] # free_nodes contient les nœuds à visiter
    free_nodes.append(t)
    closed_nodes = [] # closed_nodes contiendra les éléments déjà visités
    while (free_nodes!=[]) and (not success) and (niveaux < 100):
        # et on n'a pas atteint l'état but
        first_node = free_nodes[0] # first_node contiendra le 1er élément de free_nodes
        print()
        print('*** pas', niveaux, ': **')
        niveaux += 1
        afficher(first_node)
        free_nodes.remove(first_node) # l'élémine de cette liste et l'ajoute à closed_nodes
        closed_nodes.append(first_node)
        generated_states = transitions(first_node)#generated_states prend les états possibles à générer à partir de l'état actuel jusqu'à l'état but
        for s in generated_states:# On parcourt cette nouvelle liste
            if s == tf:# si l'un est l'état final, success devient true
                success = True
                print(' \n** pas', niveaux, ': **')
                afficher(s)
                goal_node = s # goal_node recoit l'état actuel
                free_nodes = free_nodes + generated_states# 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 placées qui représente h(x)
    free_nodes.sort(key = lambda el:(niveaux+h(el,tf))) #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
    if niveaux == 100:
        print("Recherche non conclusive")
    else:
        print(" \n --> Goal trouvé après",niveaux," iterations .")
        print('Time spent: %0.2fs' % (time.time()-start))
    print(h(t,tf),"cases mal placées depuis l'etat initial")
    aetoile(t,tf)
```

Implémentation des algorithmes (5)

- Implémentation de la fonction de recherche en profondeur limitée:

```
def recherche(t,depthFirst,star,limit):
    global end

    #On récupère la valeur du temps du système dans start_time
    start_time = time.time()

    #On initialise la liste freeNodes avec la valeur courante du taquin
    freeNodes=[t]

    #Si la valeur de limit est différente de -1 alors on initialise notre liste de niveau
    if limit != -1:
        level=[0]

    #On initialise la liste closedNodes à l'état vide
    closedNodes=[]

    #On affiche le taquin
    print("L'état Initial")
    afficher(freeNodes[0])

    #On initialise la variable check (du succès) à False
    check = False

    #On initialise le nombre d'itérations à 0
    iterations = 0
    while len(freeNodes) != 0 :
        iterations += 1
        now = freeNodes.pop(0)
        closedNodes.append(now)
        if limit != -1 :
            currentlevel=level.pop(0)
            if currentlevel >= limit :
                if estEtatFinal(now,tf):
                    check = True
                    print("\nEtat Final Atteint !")
                    afficher(now)
                    closedNodes.append(now)
                if check:
                    break
                continue
            transtionsPossibles=transitions(now)
            colander(transtionsPossibles,closedNodes)
            colander(transtionsPossibles,freeNodes)
            for i in transtionsPossibles:
                if estEtatFinal(i,tf):
                    check = True
                    print("\nEtat Final Atteint !")
                    afficher(i)
                    closedNodes.append(i)
                    break
            if check:
                break
            if depthFirst:
                freeNodes = transtionsPossibles + freeNodes
                if limit != -1:
                    level=[currentlevel+1]*len(transtionsPossibles) + level
            else:
                freeNodes = freeNodes + transtionsPossibles
                if limit != -1:
                    level=level + [currentlevel+1]*len(transtionsPossibles)
            if star:
                freeNodes.sort(key=lambda e:(iterations + h(e)))

    if check:
        print("\nThe search for the solution took %5.4f seconds \n" %(time.time() - start_time) )
        backtracking(closedNodes)
    else:
        print("The solution haven't been reached so there is no path to it, the limit is",limit)
    x=int(input("Please set the limit: "))
    recherche(t,True,False,x)
```

Conclusion

Pour clôturer, on va faire une comparaison entre les différents types de recherche. Cela aura pour but de classer ces types selon leur efficacité et de choisir le meilleur type de recherche.

L'efficacité se mesure selon le temps d'exécution, le nombre de nœuds parcourus et si l'état but est atteint ou pas.

Voici le tableau pour procéder à la comparaison

Type de parcours	Nombre de Nœuds	Temps d'exécution	Solution trouvée?
Profondeur	3	0.03 second	Oui
Largeur	9	0.03 second	Oui
A*	3	0.01 second	Oui
Profondeur limitée	4	0.0266 second	Oui

→ D'après le tableau, on conclut que le parcours A* est le meilleur parcours vu qu'il nous affiche notre solution en temps ($t = 0.01s$) minimal et avec un nombre minimal de nœuds parcourus ($n=3$).
