Project IA

Taquin

BOUFALA Yacine
KHALFOUN Rayan

Sous la direction de Mr Janodet

March 21, 2021

Contents

1 Introduction

Dans le cadre de ce devoir, il nous est demandé de créer un programme d'intelligence artificielle capable de résoudre un jeu du taquin de manière autonome. L'intelligence artificielle correspond à l'ensemble des théories et des techniques développant des programmes informatiques complexes capables de simuler certains traits de l'intelligence humaine, les jeux sont donc un excellent domaine d'application car ils constituent une véritable épreuve intellectuelle. Pour nous aider dans un notre mission nous avons utilisées des heuristiques, soit des méthodes utilisées pour fournir une solution réalisable lorsqu'un problème est trop complexe, notamment en raison de l'explosion combinatoire (grand nombre de critères à prendre en compte). Concernant le jeu en lui même, un taquin n×n est constitué de (n^2-1) carreaux à l'intérieur d'un carré pouvant contenir $n^2 jetons$: on a donc une case vide et le but est de remettre dans l'ordre les carreaux à partir d'une configuration initiale valide, une forme de puzzle. Pour le programme, nous avons décidé de mettre en place la résolution du taquin 3x3.

2 Explication theoriques

Avant de décrire le code plus en détail, nous allons commencé par voir son fonctionnement théorique. Nous avons décidé de mettre en place trois modes de résolution pour venir à bout de ce casse-tête. Dans cette partie théorique nous nous contenterons de développer uniquement la méthode A* car c'est celle demandée par le sujet

- Parcours A*

L'algorithme de recherche A*, est un parcours en largeur. Il permet de trouver une solution pour arriver d'un état initial à un état final. A* est en théorie optimisé pour nous permettre de trouver l'itinéraire le plus court pour ce chemin. A* fonctionne alors de la manière suivante : le programme commence par un noeud donné de poids 0, on applique ensuite les différentes heuristiques implémentés. Une heuristique est un bout de code nous permettant de guider notre algorithme en donnant une approximation de distance qu'il reste à parcourir. Dans le cas de notre programme, 7 heuristique sont misent en place. Une fois ce travail fait et les différentes solutions "notées", A* est chargé à l'aide d'une fonction d'evaluation défini de la sorte (f=h+g) de les trier par ordre croissant en fonction de leur poids tout les noeuds à expanser. Celui qui de moindre coûte est choisi. S'il ce dernier correspond à l'état final, l'algorithme renvoie le chemin par lequel nous sommes passé pour en arriver là, sinon l'algorithme continue en ajoutant le nouveau noeud a expanser à laliste et en triant par ordre croissant, puis en prenant le 1er élément jusqu'à trouver la solution.

3 Explication du code

Pour réaliser ce projet notre code s'organise en 6 class bien distinctes. Cette décisions de fractionner les parties s'inscrit dans une volonté de notre part de simplifier au maximum la compréhension et la clarté au moment de la lecture du code. Pour commencer, avant même la première class, nous avons importé les bibliothèques indispensables au bon fonctionnement du programme. Puis 2 variables sont créées «data» et «etat final», elles représentent respectivement l'état initiale de la grille du Taquin et la grille finale qui représente la solution. Passons maintenant à la première class : **Node**. Elle est centrale dans le fonctionnement du code puisqu'elle permet:

• La création et le parcours des noeuds : grâce à l'appel des fonctions«actions» qui permet comme son nom l'indique d'effectuer une action (ici une action entendue au sens de mouvement de pièces) et «path» qui retourne via un generateur le chemin emprunté par le programme lors du parcours récursif du graphe.

• D'évaluer la solvabilité des grilles (propriété «solved» que nous verrons plus tard). Enfin, la fonction d'évaluation : «function eval» qui utilise le résultat de l'heuristique ainsi que la distance parcourue dans le graphe c'est à dire la distance parcourue ajoutée à la distance qu'il reste potentiellement à parcourir avant d'arriver à la solution finale et qui permet de donner une mesure de l'utilité d'un noeud, de savoir si ce dernier est proche ou non de la solution.

Continuons sur la deuxième class : **Solver**. Elle regroupe trois fonctions correspondant aux différents types de parcours d'arbre possible pour la résolution du problème. Les trois fonctions on la même base, Au dessus de cette fonction de base il existe donc trois variants correspondant chacun à une manière différente de résoudre ce taquin :

• A* : prend tous les noeuds présents dans la liste queue de type «collections.deque», c'est une liste qui permet des ajout et des retrait rapide aux extrémités. Puis les tris de manière croissante suivant la fonction d'évaluation vue en amont. La fonction prend ensuite le premier de cette liste (celui avec la «F», la plus petite de l'itération courante). Elle permet dans un premier temps de vérifier si le noeud est l'état final, si oui elle retourne le chemin qui a permis d'y accéder ainsi que le nombre de noeuds vue et expansé sinon elle parcoure toute les possibilités de déplacements existant a partir de la grille courante, ensuite si le nouveau noeud ne fait pas partie de la liste des noeud vus elle l'ajoute ainsi qu'à "queue" sinon elle passe au noeud suivant .

```
if node.solved:
    return node.path

for move, action in node.actions:
    child = Node(move(), node, action)

    if child.state not in seen:
        queue.append(child)
        seen.add(child.state)
```

• Parcours en largeur : Même chose que A* sans le trie qui permet de prendre la plus petite valeur, et pas de fonction d'évaluation.

• Parcours en profondeur : Comme dit plus haut les trois méthodes ont la plus ou moins la même base, ca reprend le parcours en largeur a la difference près que l'on ne prends pas le premier élément de queue mais le dernier, ici aussi nous ne nous servons pas de la fonction d'évaluation.

La prochaine class que nous allons analyser est la class **puzzle**. Elle complète les fonctions initiées dans la class Node on y retrouve

«solved» qui prend le noeud que l'on transforme en chaine de caractére a l'aide de str puis on le compare a la solution attendu avec un (map(str, range(9))+'0') égale a «1234567890» qui est le résultat attendu.

«Action» qui permet de faire se déplacer la case vide correspondante au 0 dans notre cas dans les 4 directions possible haut, bas, droite, gauche en tenant compte des limites de la grille grâce à la condition

```
if r >= 0 and c >= 0 and r < len(self.data) and c < len(self.data) and self.data[r][c] == 0:
```

Pour se faire action s'appuis sur une deuxième fonction: «create move» qui permet d'affecter un déplacement (une permutation) de la case vide avec une autre se trouvant dans l'une des directions autorisées. c'est aussi dans cette class que nous avons définis les heuristiques, commençant par :

• Mal Placées : l'heuristique «mal place» permet de retourner le nombre de pièces qui comme son nom l'indique ne sont pas a leurs places en comparant chaque pièce avec ça position dans la grille finale, ce nombre correspond a l'approximation du nombre de coups restant avant d'atteindre la solution.

• Manhattan à poids pondérer : l'heuristique «manhattan ponderer» permet de retourner un nombre correspondant à

```
final += (abs(i - i_f) + abs(j - j_f)) * poids6[i_f][j_f]
```

ou (i, j) sont les coordonnées d'une pièce dans la grille courante et (i finale, j finale) les coordonnées de la même pièce dans la grille finale cette somme et ensuite multiplier par un poids (nous disposons comme dans l'énoncé de 6 poids) associé à chaque pièce pour enfin diviser la somme totale de toutes les pièces sur un coefficient de normalisation comme ci-dessous

```
return final // div2
```

Image est une classe uniquement réservée à l'interface graphique ,elle permet d'afficher les images correspondantes à chaque cases du tableau et permet de les faire se déplacer en fonction des différents coups joués, le tout avec une animation en temps réelle.

Melangeur est notre 5 eme class elle permet grâce trois méthodes «isSolvable»,«permutation» et «melange» de mélanger les pieces jusqu'à trouver une grille valide. On l'utilise pour relancer une partie sans avoir a quitter le programme.

Enfin nous arrivons à la fin du code et la dernière class: **Jeu**.Elle est divisée en deux parties La premiere chargée d'initié le jeu et de mettre en forme les solutions en s'aidant des fonctions présente dans solver La deuxièmes correspondant au main qui lance les parcours grâce aux touches assignées les différents mode de jeu précédemment codés

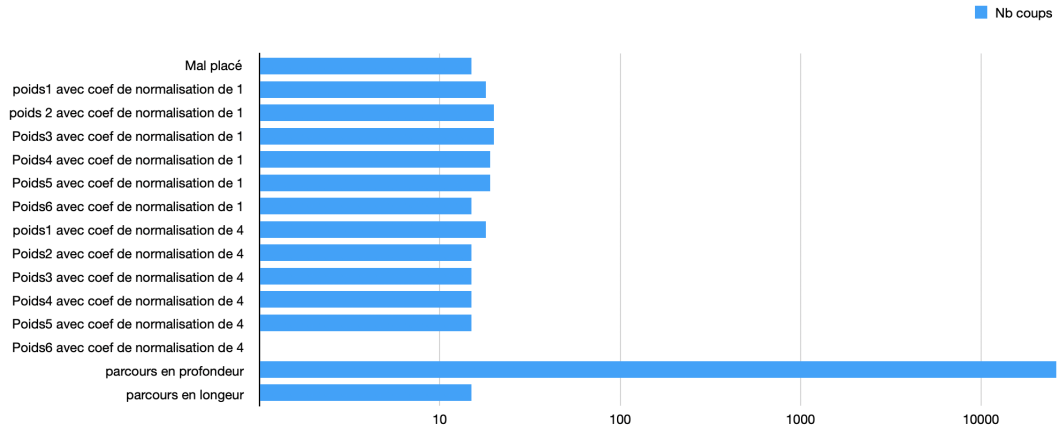
"Jeu" est donc la class bilan celle qui réunie tout les élément précédemment codé pour les faire fonctionner de cohort.

4 Test

Passons maintenant au test d'efficacité indispensable pour observer les performances des différentes heuristiques. Pour ces tests, nous avons essayé chaque stratégie sur 5 grilles différentes. Les résultats obtenus sont donc les moyennes des différents tests. Les différents critères d'évaluation que nous avons choisis de mesurer sont : le nombre de coup, le nombre de noeuds vus, le nombre de noeuds expansés et le temps de résolution.

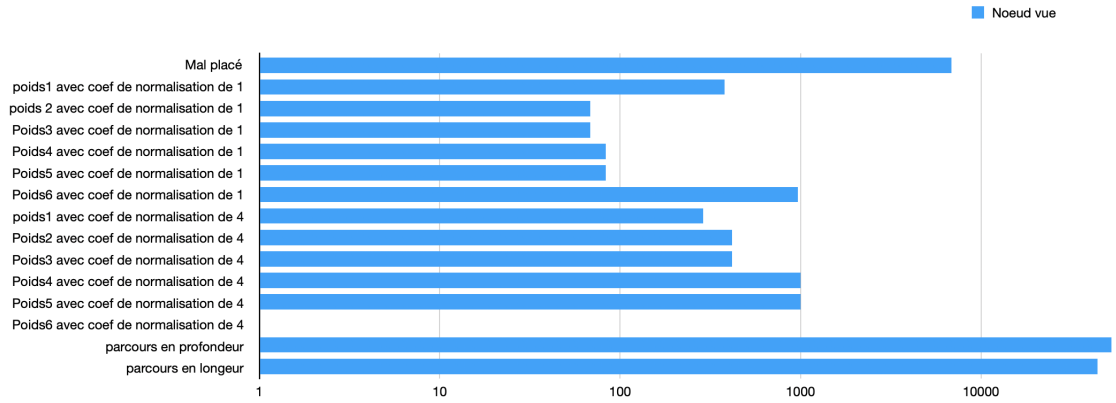
Représentation du nombre de coup en fonction des heuristiques

Commençons par le nombre de coup. Il représente le nombre de mouvement a effectué avant de résoudre le Taquin ; plus cette valeur est basse plus l'algorithme est efficace. On remarque grâce au test effectué que le nombre de coup reste plutôt stable en fonction des différentes heuristiques malgré un léger avantage pour les Manhattan avec un coefficient de normalisation de 1. Cependant on assiste à une véritable explosion de la valeur lors du parcours en profondeur, ce qui était a prévoir quand on sait que ce type de parcours n'est absolument pas optimisé pour ce genre de problème.



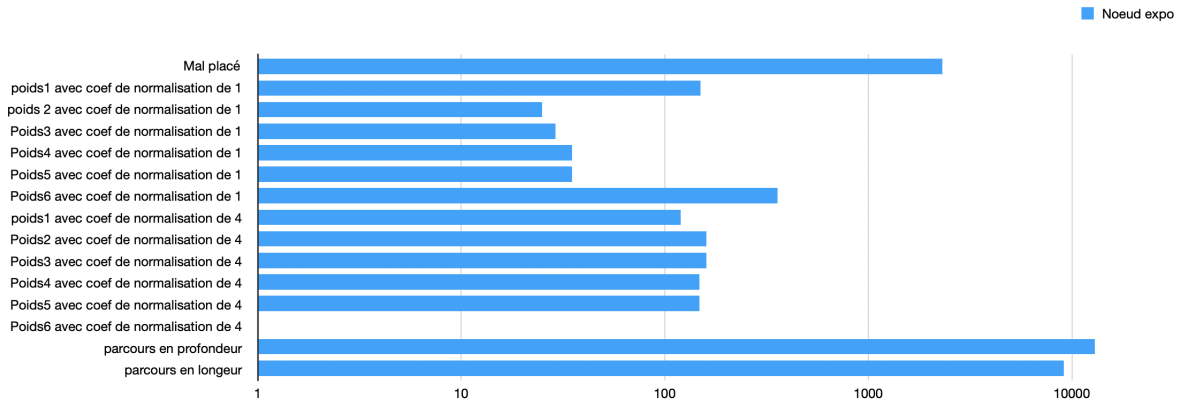
Représentation du nombre de noeuds vus

Le nombre de noeuds vus est un autre bon indicateur pour évaluer la performance de notre programme puisqu'il nous montre le nombre de noeud que compare l'algorithme avant d'arriver a la solution attendue. On peut voir que pour les Manhattan pondérés, le nombre noeuds vus est inversement proportionnel au nombre de coups joués, ce qui traduit le fait que plus le programme voit de noeuds plus il est capable de prendre une décision allant dans le sens d'une minimisation des mouvements de pièces. De part leur fonctionnement plus « hasardeux », les parcours en largeurs et en profondeur voient énormément de noeuds ce qui n'est pas vraiment conseillé dans une optique d'optimisation du temps.



Représentation du nombre de noeuds expansé

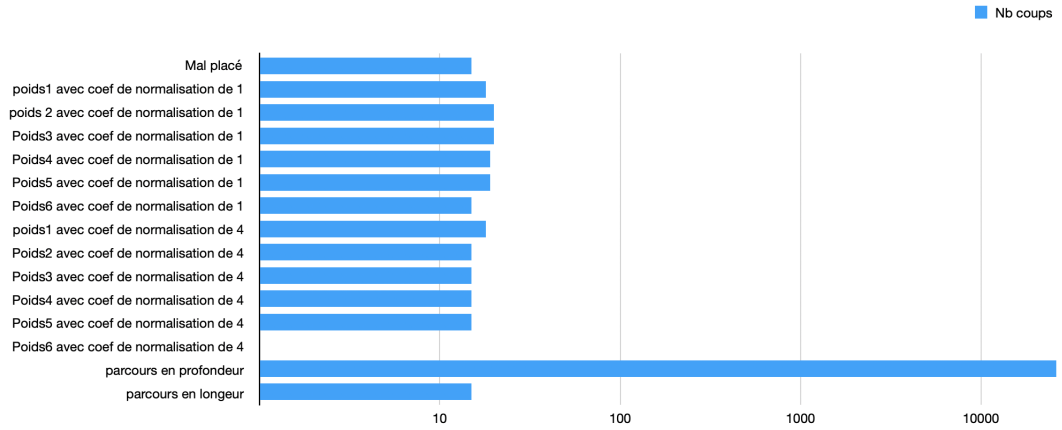
Comme pour le test précédent, tester le nombre de noeuds expansés offre un bel aperçu des performances. Ici il nous sert à vérifier le test des noeuds vus, et il s'avère en effet qu'entre les deux résultats on observe un très forte similitude se qui atteste de la cohérence des mesures.



Représentation du temps d'exécution

Pour le temps d'exécution, il va sans dire qu'il s'agit d'un facteur essentiel à prendre en compte lorsque l'on veut tester notre programme.

En ce qui concerne les autres heuristiques on peut voir que celles qui voient et expansent le plus de noeuds sont aussi celles qui prennent le moins de temps. Mal placé met plusieurs dizaines de secondes avant de terminer contre 0,001 seconde pour certaines autres heuristiques beaucoup moins gourmandes en noeuds. Mais, comme vu précédemment, plus un algorithme expanse et voit de possibilités différentes, plus il est efficace pour trouver une solution minimale en nombre de coups.



Représentation du temps d'exécution

Pour conclure au sujet des statistiques et des tests, nous pouvons voir qu'en fonction des différentes heuristiques utilisées les résultats changent du tout au tout. Malgré cela, ces différences reste explicables et logiques c'est donc à nous d'équilibrer au mieux le programme en fonction des leviers qui sont à notre disposition afin d'obtenir le meilleur résultat en gardant un temps d'exécution raisonnable en limitant le nombre de noeuds vus et expansés.

5 Extention

Nous avons essayé, afin de répondre aux besoins de ce devoir, le plus de fonctionnalités possibles. Cependant, il reste des pistes a exploré comme la mise en place d'un mode de jeu « manuel » ou d'un mode « versus » contre la machine. Il nous est aussi possible d'ajouter d'autres mesures de grilles puisqu'ici seul le taquin 3x3 est pris en compte mais il est possible d'étendre à 4x4, 5x5 etc. Voilà selon nous les possibilités d'extension possibles pour avoir un programme complet qui permet à la fois la résolution du casse-tête mais aussi un moyen pour l'utilisateur de s'amuser.

6 Conclusion

Ainsi, la réalisation de ce programme nous a permis d'en apprendre davantage sur l'intelligence artificielle et plus précisément les heuristiques. En effet, ce concept et dans un cadre plus large, l'algorithme A*, que nous voyons pour la première fois, est indispensable à cette discipline en ce qu'il permet de réduire grandement et d'améliorer les algorithmes de recherche de chemin. C'est donc un exercice qui nous a été très formateur et nous servira dans la suite de notre cursus. De plus, chaque travail de groupe nous permet de nous familiariser davantage avec l'environnement dans lequel il nous sera demandé d'évoluer à l'avenir, et d'apprendre à nous organiser de plus en plus dans la gestion des tâches dans le cadre de gros projets.

7 Code

```
import pygame
from itertools import product
import collections
from time import sleep, perf_counter
from random import sample
from pprint import pprint

poids1 = [[36, 12, 12], [4, 1, 1],[4, 1, 0]]
poids2 = [[8, 7, 6], [5, 4, 3], [2, 1, 0]]
poids3 = poids2
poids4 = [[8, 7, 6], [5, 3, 2], [4, 1, 0]]
poids5 = poids4
poids6 = [[1, 1, 1], [1, 1, 1],[ 1, 1, 0]]

div1 = div3 = div5 = 4
div2 = div4 = div6 = 1

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

size = (600, 800)[len(data) == 4]

class Node:

    def __init__(self, puzzle, parent=None, action=None):
        self.puzzle = puzzle
        self.parent = parent
        self.action = action

        if (self.parent != None):
            self.g = parent.g + 1
        else:
            self.g = 0

    @property
    def state(self):
        return str(self)
```

```
    @property
    def path(self):
        node, p = self, []
        while node:
            p.append(node)
            node = node.parent
        yield from reversed(p)

    @property
    def solved(self):
        return self.puzzle.solved # bool

    @property
    def actions(self):
        return self.puzzle.actions

    @property
    def heuristique(self):
        #return self.puzzle.mal_place
        return self.puzzle.manhattan_ponderer

    @property
    def function_eval(self):
        return self.heuristique + self.g

    def __str__(self):
        return str(self.puzzle)

class Solver:

    def __init__(self, start):
        self.start = start

    def solve_Astar(self):
        queue = collections.deque([Node(self.start)])
        seen = set()
        seen.add(queue[0].state)
        while queue:
            queue = collections.deque(sorted(list(queue),
            key = lambda node: node.function_eval))
            node = queue.popleft()

            if node.solved:
                return node.path, len(seen), len(queue)
```

```
            for move, action in node.actions:
                child = Node(move(), node, action)

                if child.state not in seen:
                    queue.append(child)
                    seen.add(child.state)

    def solve_profondeur(self):
        queue = collections.deque([Node(self.start)])
        seen = set()
        seen.add(queue[0].state)
        while queue:
            node = queue.pop()

            if node.solved:
                return node.path, len(seen), len(queue)

            for move, action in node.actions:
                child = Node(move(), node, action)
                if child.state not in seen:
                    queue.append(child)
                    seen.add(child.state)

    def solve_largeur(self):
        queue = collections.deque([Node(self.start)])
        seen = set()
        seen.add(queue[0].state)
        while queue:
            node = queue.popleft()

            if node.solved:
                return node.path, len(seen), len(queue)

            for move, action in node.actions:
                child = Node(move(), node, action)
                if child.state not in seen:
                    queue.append(child)
                    seen.add(child.state)

class Puzzle:

    def __init__(self, data, etat_final=etat_final):
        self.data = data
        self.etat_final = etat_final
```

```
@property
def solved(self):
    return str(self) == ''.join
    (map(str, range(1, pow(len(self.data), 2)))) + '0'

@property
def actions(self):
    def create_move(at, to):
        return lambda: self._move(at, to)

    moves = []
    for i, j in product(range(len(self.data)), range(len(self.data))):
        directions = {'droite': (i, j-1),'gauche': (i, j+1), 'bas':
        (i-1, j),'haut': (i+1, j)}
        for action, (r, c) in directions.items():
            if r >= 0 and c >= 0 and r < len(self.data) and
            c < len(self.data) and self.data[r][c] == 0:
                move = create_move((i,j), (r,c)), action
                moves.append(move)
    return moves

@property
def mal_place (self):
    piecesMalPlace = 0
    for i in range(len(self.data)):
        for j in range(len(self.data)):
            if self.data[i][j] != self.etat_final[i][j]:
                piecesMalPlace += 1
    return piecesMalPlace   # valeur

def getPosition_x_y(self, board, num):
    for i in range(len(self.data)):
        for j in range(len(self.data)):
            if board[i][j] == num:
                return i, j

# poids pour les peices de la 1er ligne
@property
def manhattan_ponderer(self):
    final = 0;
    for x in range(len(self.data)):
        for y in range(len(self.data)):
            i , j = self.getPosition_x_y(self.data, self.data[x][y])
            i_f, j_f = self.getPosition_x_y
            (self.etat_final, self.data[x][y])
```

```
                final += (abs(i - i_f) + abs(j - j_f)) * poids6[i_f][j_f]
        return final // div1

    def copy(self):
        data = []
        for row in self.data:
            data.append([x for x in row])
        return Puzzle(data)

    def _move(self, at, to):
        copy = self.copy()
        i, j = at
        r, c = to
        copy.data[i][j], copy.data[r][c] =
        copy.data[r][c], copy.data[i][j]
        return copy

    def pprint(self):
        for row in self.data:
            print(row)
        print()

    def __str__(self):
        return ''.join(map(str, self))

    def __iter__(self):
       for row in self.data:
            yield from row

class Images :

    def __init__(self, data):
        self.ecran = pygame.display.set_mode((size, size))
        pygame.display.set_caption('Le Taquin __Projet - IA__')
        pygame.display.set_icon(pygame.image.load("image/taquin.png"))
        self.data = data

    def loadImages(self):

        "afficher les images sur le canvas"
        # 1er palier
        position1 = pygame.image.load
        ("image/"+str(self.data[0][0])+".png")
        self.ecran.blit(position1, (24, 24))
        position2 = pygame.image.load
```

```
("image/"+str(self.data[0][1])+".png")
self.ecran.blit(position2, (225, 24))
position3 = pygame.image.load
("image/"+str(self.data[0][2])+".png")
self.ecran.blit(position3, (426, 24))

if(len(self.data) == 4):
      position4 = pygame.image.load
      ("image/"+str(self.data[0][3])+".png")
      self.ecran.blit(position4, (626, 24))

# 2nd palier
position5 = pygame.image.load
("image/"+str(self.data[1][0])+".png")
self.ecran.blit(position5, (24, 225))
position6 = pygame.image.load
("image/"+str(self.data[1][1])+".png")
self.ecran.blit(position6, (225, 225))
position7 = pygame.image.load
("image/"+str(self.data[1][2])+".png")
self.ecran.blit(position7, (426, 225))

if(len(self.data) == 4):
      position8 = pygame.image.load
      ("image/"+str(self.data[1][3])+".png")
      self.ecran.blit(position8, (626, 224))

# 3 palier
position9 = pygame.image.load
("image/"+str(self.data[2][0])+".png")
self.ecran.blit(position9, (24, 426))
position10 = pygame.image.load
("image/"+str(self.data[2][1])+".png")
self.ecran.blit(position10, (225, 426))
position11 = pygame.image.load
("image/"+str(self.data[2][2])+".png")
self.ecran.blit(position11, (426, 426))

if(len(self.data) == 4):
     position12 = pygame.image.load
     ("image/"+str(self.data[2][3])+".png")
     self.ecran.blit(position12, (626, 424))

    # 4 palier
     position13 = pygame.image.load
```

```
            ("image/"+str(self.data[3][0])+".png")
            self.ecran.blit(position13, (24, 626))
            position14 = pygame.image.load
            ("image/"+str(self.data[3][1])+".png")
            self.ecran.blit(position14, (225, 626))
            position15 = pygame.image.load
            ("image/"+str(self.data[3][2])+".png")
            self.ecran.blit(position15, (426, 626))
            position16 = pygame.image.load
            ("image/"+str(self.data[3][3])+".png")
            self.ecran.blit(position16, (626, 626))

    def fill(self, color):
        self.ecran.fill(color)

    def update(self, new_value, time=0.3):
        self.data = new_value
        self.loadImages()
        pygame.display.flip()
        sleep(time)


class Melangeur:

    def __init__(self, data):
        self.data = data

    @property
    def isSolvable(self) :
        inv_count = 0
        for i in range(0, len(self.data)-1) :
            for j in range(i + 1, len(self.data)) :
                if (self.data[j][i] > 0 and self.data[j][i] >
                self.data[i][j]) :
                    inv_count += 1
        return (inv_count % 2 == 0)

    def permutation(self, source, source1, destination, destination1):
        tmp = self.data[source][source1]
        self.data[source][source1] = self.data[destination][destination1]
        self.data[destination][destination1] = tmp

    @property
    def melange(self):
        for i in range(10):
```

```
            l = sample(range(len(self.data)), len(self.data)-1)
            l1 = sample(range(len(self.data)), len(self.data)-1)
            self.permutation(l[0], l1[0], l[1], l1[1])
        return self.data

    @property
    def getSolvable(self):
        while self.isSolvable :
            self.melange

class Jeu :

    global solver

    def __init__(self, data=data):
        self.data = data
        self.play = True
        self.exec = Melangeur(self.data)
        self.images = Images(self.data)

    def Astar(self):
        solver = Solver(Puzzle(self.data))
        begin = perf_counter()
        p, v, e = solver.solve_Astar()
        end = perf_counter()
        compteur = 0
        if p != None:
            for node in p:
                print("coup {}:".format(compteur))
                node.puzzle.pprint()
                self.images.update(node.puzzle.data, 0.1)
                # self.data = node.puzzle.data
                compteur += 1
            print("Solution en : " + str(compteur-1)+ " coups")
            print("Total des noeuds vue: " + str(v))
            print("Total des noeuds expense: " + str(e))
            print("Temps : " + str(end - begin) + " seconde(s)")
        else:
            pprint("Erreur inconnue, veuillez redemarer !")

    def profondeur(self):
        solver = Solver(Puzzle(self.data))
        begin = perf_counter()
        p, v, e = solver.solve_profondeur()
        end = perf_counter()
```

```
        compteur = 0
        if p != None:
            for node in p:
                print("coup {}:".format(compteur))
                node.puzzle.pprint()
                compteur += 1
            print("Solution en : " + str(compteur-1)+ " coups")
            print("Total des noeuds vue: " + str(v))
            print("Total des noeuds expense: " + str(e))
            print("Temps : " + str(end - begin) + " seconde(s)")
        else:
            pprint("Erreur inconnue, veuillez redemarer !")

    def longeur(self):
        solver = Solver(Puzzle(self.data))
        begin = perf_counter()
        p, v, e = solver.solve_largeur()
        end = perf_counter()
        compteur = 0
        if p != None:
            for node in p:
                print("coup {}:".format(compteur))
                node.puzzle.pprint()
                self.images.update(node.puzzle.data, 0.1)
                # self.data = node.puzzle.data
                compteur += 1
            print("Solution en : " + str(compteur-1)+ " coups")
            print("Total des noeuds vue: " + str(v))
            print("Total des noeuds expense: " + str(e))
            print("Temps : " + str(end - begin) + " seconde(s)")
        else:
            pprint("Erreur inconnue, veuillez redemarer !")

    def main(self):
        while self.play:
            for even in pygame.event.get():
                if even.type == pygame.QUIT:
                    self.play = False

                if even.type == pygame.KEYDOWN:
                    if even.key == pygame.K_c:
                        self.data = self.exec.melange

                        if(not self.exec.isSolvable): # si non solvable
                            self.exec.getSolvable
```

```
                    self.images.update(self.data, 0)
                    pygame.display.flip()
                    pprint(self.exec.data)

                if even.key == pygame.K_SPACE:
                    pprint("--------r solution avec
                    A* ----------")
                    self.Astar()

                if even.key == pygame.K_p:
                    pprint("-------- r solution avec
                    recherche en profondeur ----------")
                    self.profondeur()

                if even.key == pygame.K_l:
                    pprint("-------- r solution avec
                    recherche en largeur ----------")
                    self.longeur()

                if even.key == pygame.K_q:
                    self.start = False

            self.images.fill((0, 0, 0))
            self.images.loadImages()
            pygame.display.flip()

if __name__ == '__main__':
    pygame.init()
    Jeu().main()
    pygame.quit()
```