Université d'Evry (2022/2023), **IA**, 1–17

MINI PROJET IA

L3 Informatique (CILS)

Camille Berthaud – 20202238

Projet encadré par Jean-Christophe Janodet

Introduction

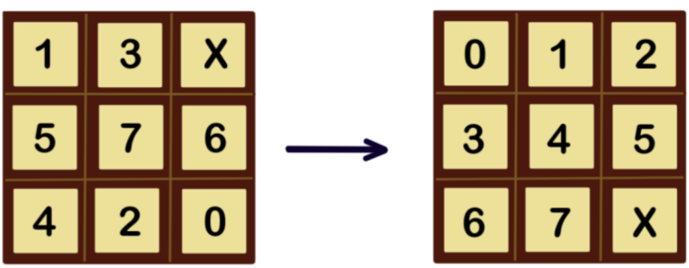
Abstract

Dans le cadre de la matière d'Intelligence Artificielle, nous avons eu pour consigne de développer un programme capable de résoudre automatiquement un jeu de taquin. Le domaine de l'IA englobe toutes les théories et techniques nécessaires pour créer des programmes informatiques complexes capables de simuler certaines capacités humaines, ce qui inclut les jeux en tant que domaine d'application idéal pour relever des défis intellectuels. Pour répondre à ce défi, nous avons utilisé des heuristiques, des méthodes fournissant des solutions réalisables pour des problèmes complexes.

1. Etude théorique

1.1 Le problème du taquin

Un taquin est un puzzle carré contenant $(n^2 - 1)$ carreaux et une case vide (représentée par un "X" sur les schémas ci-dessous). L'objectif est de reconstituer l'ordre des carreaux à partir d'une configuration initiale valide. Nous avons choisi de mettre en oeuvre la résolution d'un taquin de taille 3×3



Dans le jeu de taquin, il est possible de déplacer les tuiles uniquement en les faisant glisser dans la case vide, et ce déplacement ne peut se faire que pour une seule tuile à la fois. A travers ce projet nous allons déterminer quelle est la séquence minimale de déplacements à réaliser pour obtenir la solution. Pour ce faire, nous avons décidé d'utiliser l'algorithme A-star ou A*.

1.2 L'Algorithme A*

L'algorithme A* est un algorithme de recherche de chemin utilisé pour trouver le chemin le plus court entre un nœud de départ et un nœud d'arrivé dans un graphe pondéré. Il utilise une approche heuristique pour améliorer l'efficacité de la recherche en explorant d'abord les nœuds les plus prometteurs.

L'algorithme fonctionne en utilisant une fonction d'évaluation f(n) qui estime le coût total pour atteindre le nœud final en passant par le nœud actuel. Cette fonction est définie comme la somme de deux autres fonctions :

$$f(n) = g(n) + h(n)$$

Avec :

- $g(n)$: le coût réel pour atteindre le nœud n depuis le nœud initial
- $h(n)$: une estimation heuristique du coût restant pour atteindre le nœud final depuis le nœud n
- $f(n)$: coup total du noeud n

La fonction d'évaluation f(n) est utilisée pour classer les nœuds à explorer en priorité. Ainsi, les nœuds avec les plus petites valeurs de f(n) sont explorés en premier. Cela permet de rechercher en premier les nœuds les plus prometteurs, c'est-à-dire ceux qui ont le potentiel d'atteindre le nœud final avec le coût le plus faible.

L'algorithme A* garantit de trouver le chemin le plus court si la fonction heuristique est admissible, c'est-à-dire si elle est toujours inférieure ou égale au coût réel pour atteindre le nœud final. Cependant, l'algorithme peut être coûteux en temps de calcul si le graphe est très grand et complexe, ou si la fonction heuristique n'est pas bien choisie.

2. Explication du code

Pour la réalisation du projet, le code s'organise en 3 classes distinctes : Taquin, Algo et Test. Le choix de fractionner le code de cette manière permet de clarifier le code, et surtout de rendre générique la classe Algo, afin qu'elle puisse résoudre tout type de problème.

2.1 Class Taquin

2.2 class Taquin

constructeur : définition d'un état

Le constructeur de la classe Taquin a un paramètre qui peut avoir 2 types :
- une chaîne de caractères qui représente un état du taquin. Dans ce cas, le constructeur vérifie la longueur de la chaîne. Si la condition est respectée, la taille de la grille est calculée, ainsi que la distance f jusqu'à l'état final et l'historique des mouvements effectués pour arriver à cet état.
- un objet de type Taquin. Dans ce cas, le constructeur est un constructeur de copie et sert à générer un enfant à partir d'un état Taquin parent. On conserve alors l'état, la taille, l'historique et le coût final pour arriver jusqu'à la solution.

```python
    def __init__(self, param):

        if isinstance(param, str):
            self.state = param
            self.size = int(math.sqrt(len(self.state)))
            if self.size ** 2 != len(self.state):
                raise Exception("Le taquin doit être un carre")
            self.f = 0
            self.history = ''
        elif isinstance(param, Taquin):
            self.state = param.state
            self.size = param.size
            self.history = param.history
            self.f = param.f
        else:
            raise Exception("Parametre inconnu")
```

projet2.py

move() et getChild() : obtention des enfants d'un état

Pour résoudre le problème, il est essentiel de définir les états et les actions associées. Une action correspond à un déplacement de tuile impliquant l'inversion de sa position avec celle de la case 'X'. Seules les tuiles adjacentes à la case vide peuvent être déplacées, et une tuile déplacée ne peut pas être replacée à sa position précédente pour optimiser la résolution du problème.

À partir de ce postulat, on peut en déduire les conclusions suivantes :

- Si la case vide est située à l'intérieur du puzzle à l'état initial, il y a quatre coups possibles.
- Si la case vide est située sur un bord du puzzle (mais pas dans un coin) à l'état initial, il y a trois coups possibles
- Si la case vide est située dans un coin du puzzle à l'état initial, il y a deux coups possibles

La méthode move() prend en paramètre une direction ('N' : Nord, 'S' : Sud, 'O' : Ouest, 'E' : Est) représentant la direction dans laquelle la tuile doit être déplacée. La méthode commence par trouver la position du carreau 'X' dans l'état actuel du taquin.

La méthode calcule ensuite la nouvelle position de la tuile 'X' en fonction du paramètre direction. Si la nouvelle position est hors limites (au-delà des bords du taquin), la méthode retourne None.

Si la nouvelle position est valide, la méthode crée une nouvelle instance de la classe Taquin appelée res, qui est une copie de l'état actuel du taquin. Elle échange ensuite les positions de la tuile 'X' et de la tuile à la nouvelle position dans l'état enfant qui vient d'être créé. Enfin, elle ajoute le paramètre direction à l'historique du taquin et retourne le nouvel état du taquin.

La méthode getChild() de la classe "Taquin" essaie de faire tous les déplacements possibles et ne retourne que les solutions valides (différentes de None) sous la forme d'une liste d'objets Taquin.

```python
    def move(self, direction):

        pos = self.state.find('X')
        newpos = pos
        if pos is None:
            return None

        if (direction == "N"):
            newpos = pos-3
            if (newpos < 0):
                return None
        elif (direction == "S"):
            newpos = pos+3
            if (newpos > (self.size*self.size)-1):
                return None
        elif (direction == "O"):
            if (pos % 3 == 0):
                return None
            newpos = pos-1
        elif (direction == "E"):
            if (pos % 3 == 2):
                return None
            newpos = pos+1

        res = Taquin(self)
        tmp = res.state[newpos]
        res.state = res.state.replace('X', 'Y')
        res.state = res.state.replace(self.state[newpos], 'X')
        res.state = res.state.replace('Y', tmp)
        res.history += direction
        return res

    def getChild(self):
        children = []

        if self.move("N") != None:
            children.append(self.move("N"))
```

```python

        if self.move("S") != None:
            children.append(self.move("S"))

        if self.move("E") != None:
            children.append(self.move("E"))

        if self.move("O") != None:
            children.append(self.move("O"))

        return children
```

projet2.py

dist_manhattan() et get_h_score() : calculs des heuristiques

Lors de l'utilisation de l'algorithme A*, il est essentiel de sélectionner des heuristiques appropriées. Nous avons choisi d'utiliser 6 heuristiques qui surestiment la distance nécessaire pour atteindre l'état final. Les six heuristiques sont basées sur la distance de Manhattan et sont identifiées comme h1, h2, h3, h4, h5 et h6. Chacune de ces heuristiques est calculée avec des pondérations différentes en fonction des tuiles.

Tuile	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**X**
π_1	36	12	12	4	1	1	4	1	0
$\pi_2 = \pi_3$	8	7	6	5	4	3	2	1	0
$\pi_4 = \pi_5$	8	7	6	5	3	2	4	1	0
π_6	1	1	1	1	1	1	1	1	0

Avec pour coefficient de normalisation respectifs :

```python
COEFF = (4, 1, 4, 1, 4, 1)  # rho1 a rho6
```

projet2.py

A partir de ces pondérations nous pouvons calculer la distance de Manhattan :

$$h_k(E) = \left(\sum_{i=1}^{8} \pi_k(i) \times \varepsilon_E(i) \right) \quad // \quad \rho_k$$

$\varepsilon_E(i)$ correspond au nombre de déplacements nécessaires pour déplacer la tuile i de l'état initial à l'état final, calculé à partir de cette formule :

$$d(A, B) = |X_B - X_A| + |Y_B - Y_A|$$

La méthode dist_manhattan() calcule la distance de Manhattan pondérée entre une tuile spécifiée et sa position finale. La méthode get_h_score() calcule le score heuristique pour l'état actuel en utilisant la distance de Manhattan pondérée pour toutes les tuiles, en fonction du coefficient de normalisation et du poids associé à chaque tuile.

```
    def dist_manhattan(self, e, state_final):
        # renvoie le nombre de deplacement a faire pour obtenir l'etat final
        pos = self.state.find(e)

        # cherche la position de la tuile dans l'etat final
        pos_final = state_final.state.index(e)
        # deplacements a faire pour obtenir la position finale de la tuile
        h = abs(pos % self.size - pos_final % self.size) + \
            abs(pos // self.size - pos_final // self.size)
        return h

    def get_h_score(self, mod, final_state):
        res = 0
        a = 0
        for e in self.state:
            if (e == "X"):
                continue

            a += POIDS[mod-1][int(e)] * (self.dist_manhattan(e, final_state))
            res = a // COEFF[mod-1]
        return res
```

projet2.py

2.3 class Algo

A_star() : Algorithme A*

Cette classe implémente l'algorithme A*. La méthode A_star() prend en entrée un mode de recherche (heuristique), l'état initial et l'état final. Cette méthode retourne l'état final ainsi que le chemin parcouru pour l'atteindre.

La méthode A_star() utilise une liste ordonnée de l'ensemble frontière border_set pour stocker les états à explorer, un dictionnaire de l'ensemble frontière border_dict pour référencer chaque état de la frontière, et un dictionnaire des états explorés explored_dict pour éviter de visiter les mêmes états plusieurs fois.

A_star() parcourt la frontière en extrayant l'état avec la plus petite valeur de la fonction d'évaluation f. Elle explore ensuite les enfants de l'état extrait, en calculant leur fonction d'évaluation f et en les ajoutant à la frontière s'ils ne sont pas déjà présents. Elle met également à jour la frontière et le dictionnaire si un enfant est déjà présent avec un chemin plus court. La boucle se poursuit jusqu'à ce que l'état final soit atteint ou que la frontière soit vide. Si l'état final n'est pas atteint, elle renvoie None.

```
    def find(self, mode, initial_state, final_state):
        border_set = []  # liste ordonnee de l'ensemble fontiere
        border_dict = {}  # dictionnaire de l'ensemble frontiere
        explored_dict = {}  # savoir si l'etat a deja ete parcouru
        cpt = 0
```

```python
    # au debut mettre l'etat initial dans frontiere + explored
    initial_state.f = initial_state.get_h_score(mode, final_state)
    border_set.append(initial_state)
    border_dict[initial_state] = initial_state

    while border_set:
        current = border_set.pop(0)

        # supprime de border_dict et ajoute a explored_dict
        del border_dict[current]
        explored_dict[current] = current
        self.explored_count += 1

        # Verifier si l'etat courant de la liste correspond a l'etat final
        if current.state == final_state.state:
            # Si oui, terminer la boucle
            return current

        # recuperer les enfants de ce premier etat de la liste fontiere
        neighbors = current.getChild()

        for neighbor in neighbors:
            # calculer les heuristiques des enfants
            neighbor.f = neighbor.get_traveledPath() + neighbor.get_h_score(mode, final_state)

            # rechercher dans l ensemble si les etats des enfants ne sont pas deja presents => Si l etat est deja present et a un chemin plus long que le nouveau, on le supprime de l'ensemble et du dictionnaire frontiere
            if neighbor in border_dict:
                found = border_dict[neighbor]
                if (found.get_traveledPath() > neighbor.get_traveledPath()):
                    border_set.remove(found)
                    del border_dict[found]
                else:
                    continue

            # Si l etat est deja present et a un chemin plus court que le nouveau, on oublie le nouveau
            # sinon on l insere dans l ensemble fontiere par ordre d heuristique croissante, et on le reference dans le dictionnaire fontiere
            if neighbor in explored_dict:
                found = explored_dict[neighbor]

                if (found.get_traveledPath() > neighbor.get_traveledPath()):
                    explored_dict[neighbor] = neighbor
                else:
                    continue

            # Ajouter le voisin a la frontiere en triant par ordre croissan
```

```
                border_set.append(neighbor)
                border_set.sort(key=lambda x: x.f)
                border_dict[neighbor] = neighbor

        # Si on n a pas trouve d etat final, on renvoie None
        return None
```

projet2.py

2.4 class Test

Le jeu du taquin n'a pas forcément de solution suivant l'état initial et l'état final choisis. Il faut notamment que le nombre de différences de positionnement des tuiles, entre ces deux états, soit pair.
Nous avons choisi d'utiliser cette propriété pour éviter de générer des cas insolubles lors de la phase de tests. Pour cela, nous nous sommes également placés dans un cas particulier où l'état final souhaité sera le suivant : "01234567X".

isSolvable() : vérifier si le taquin est résolvable

La méthode isSolvable() calcule le nombre d'inversions dans le taquin, entre l'état initial et l'état final. Le taquin est résolvable uniquement si le nombre d'inversions est pair.

```
    def isSolvable(self):
        # Convertir la chaîne de caractères en une liste d'entiers
        nums = [int(x) for x in self.data if x != 'X']

        # Compter le nombre d'inversions dans la liste
        inversions = 0
        for i in range(len(nums)):
            for j in range(i+1, len(nums)):
                if nums[i] > nums[j]:
                    inversions += 1

        # Si le nombre d'inversions est pair, le puzzle est soluble
        return inversions % 2 == 0
```

projet2.py

melanger_chaine() : mélanger le taquin

La méthode melanger_chaine() mélange aléatoirement les caractères d'une chaîne, elle renvoie la chaîne obtenue. Cette méthode utilise la fonction shuffle() du module random de Python pour mélanger les caractères de la chaîne.

```
    def melanger_chaine(self):
        """Mélange une chaîne de caractères"""
        list_car = list(self.data)
        random.shuffle(list_car)
        chaine_melangee = ''.join(list_car)
        return chaine_melangee
```

projet2.py

3. Tests de performance

Afin d'évaluer les performances des différentes heuristiques, nous avons réalisé quelques tests.

Les critères d'évaluation que nous avons choisis de mesurer sont :
– le temps de résolution : le temps nécessaire à l'algorithme pour effectuer les calculs et retourner un résultat
– le nombre de nœuds explorés : le nombre de nœuds examinés avant d'arriver à la solution souhaitée. Ce critère permet d'évaluer l'espace mémoire utilisé par l'algorithme.
– le nombre de coup : le nombre déplacements minimum que l'algorithme a trouvé pour parvenir à la solution.

3.1 Test de performances en fonction des différentes heuristiques pour un état initial donné

Etat initial : taquin = Taquin("135X76420")

Voici les résultats :

	Temps de résolution	Nombre de noeuds explorés	Nombre de coups
h1	0.04266s	738	31
h2	0.00804s	196	39
h3	0.83842s	4737	27
h4	0.01186s	252	39
h5	0.76512s	4293	27
h6	0.34134s	2835	25

Nous pouvons voir que chaque heuristique permet d'arriver à la solution avec un nombre de coups différent. Nous pouvons constater aussi que plus le nombre de noeuds explorés est faible, plus le nombre de coups est important.

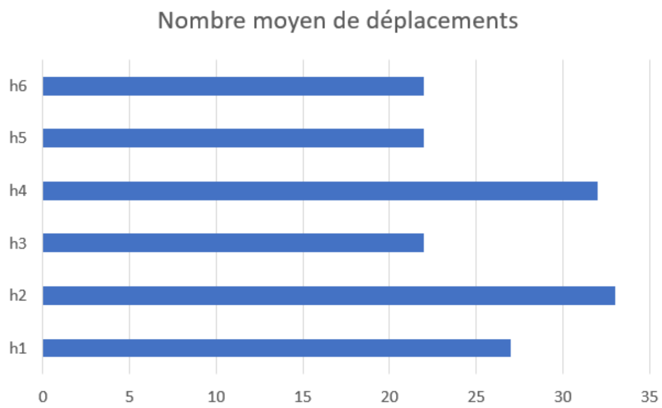
Parmi les heuristiques, il semblerait que h6 soit la plus intéressante :
– le nombre de coups est optimal
– le temps de résolution et le nombre de noeuds explorés sont dans la moyenne.

Les heuristiques h3 et h5 permettent également d'avoir une solution la plus rapide en nombre de coups, en revanche elles sont moins intéressantes en terme de temps d'exécution et d'utilisation de la mémoire.

3.2 Test de performances pour des états initiaux générés aléatoirement en fonction des différentes heuristiques

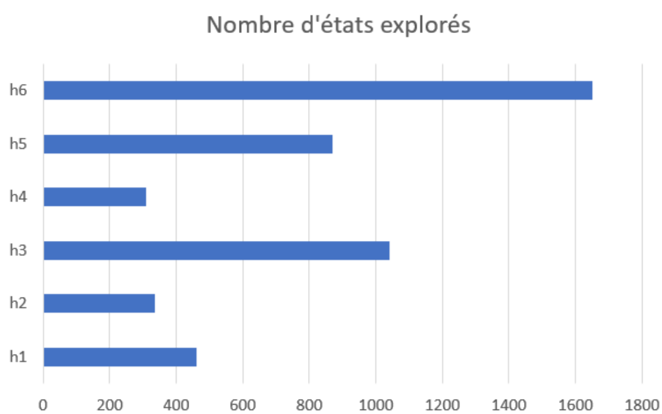
Nous allons maintenant vérifier si le test précédent est représentatif, en testant sur un échantillon de 1000 taquins générés aléatoirement. Les résultats obtenus sont les moyennes des différents tests. Dans les tests réalisés en moyenne 50 % des taquins sont résolvables.

Le nombre de coups



Nous constatons que le résultat est cohérent avec le test est précédent. Les heuristiques h3, h5, h6 permettent de parvenir à la solution avec un nombre de coup le plus faible.

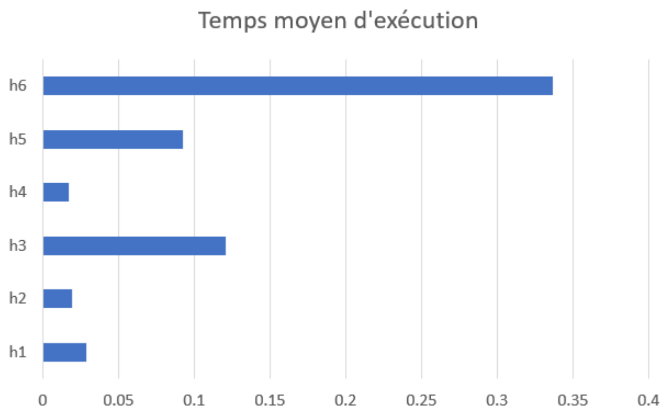
Le nombre d'états explorés



Contrairement au test précédent, l'heuristique h6 présente une consommation beaucoup plus importante en terme de mémoire, dû à un nombre d'états explorés élevé.

Nous pouvons constater que le nombre de nœuds explorés est inversement proportionnel au nombre de coups joués. Celà suggère que plus le programme explore de nœuds, plus il est en mesure de trouver une solution qui minimise le nombre de mouvements de pièces.

Le temps d'exécution



Nous pouvons constater que les heuristiques qui expandent le plus de noeuds sont les moins rapides. En revanche un algorithme explore des possibilités, plus il est susceptible de trouver une solution optimale en nombre de coups.

Sur un échantillon de 1000 tests, nous constatons que h5 est l'heuristique la plus intéressante, elle est la plus optimale et présente une consommation en espace mémoire et un temps d'exécution les plus faibles en moyenne.

4. Extensions possibles

5. Conclusion

En conclusion, l'algorithme A* est un algorithme complet, capable de trouver une solution à un problème complexe si elle existe. Il utilise une heuristique pour guider la recherche, ce qui permet de réduire considérablement le nombre de nœuds explorés par rapport à des recherches simples de type BFS ou DFS. En revanche, la difficulté de cet algorithme réside dans le choix de l'heuristique. Nous avons vu dans les exemples précédents qu'un réglage de la pondération était nécessaire pour arriver à une solution acceptable en terme de consommation de mémoire et de temps d'exécution.

6. Annexes

```


from gettext import find
import math
```

```python
import random
from collections import deque
import time

POIDS = ((36, 12, 12, 4, 1, 1, 4, 1, 0),  # pi1
         (8, 7, 6, 5, 4, 3, 2, 1, 0),  # pi2 = pi3
         (8, 7, 6, 5, 4, 3, 2, 1, 0),  # pi3 = pi2
         (8, 7, 6, 5, 3, 2, 4, 1, 0),  # pi4 = pi5
         (8, 7, 6, 5, 3, 2, 4, 1, 0),  # pi5 = pi4
         (1, 1, 1, 1, 1, 1, 1, 1, 0))  # pi6

COEFF = (4, 1, 4, 1, 4, 1)  # rho1 a rho6


class Taquin:

    def __init__(self, param):

        if isinstance(param, str):
            self.state = param
            self.size = int(math.sqrt(len(self.state)))
            if self.size ** 2 != len(self.state):
                raise Exception("Le taquin doit être un carre")
            self.f = 0
            self.history = ''
        elif isinstance(param, Taquin):
            self.state = param.state
            self.size = param.size
            self.history = param.history
            self.f = param.f
        else:
            raise Exception("Parametre inconnu")

    def __eq__(self, other):
        return isinstance(other, Taquin) and self.state == other.state

    def __hash__(self):
        return hash(self.state)

    def __str__(self):
        grille = [self.state[i * self.size:(i + 1) * self.size]
                  for i in range(self.size)]
        return '\n'.join(' '.join(str(cell) for cell in row) for row in grille)

    def move(self, direction):

        pos = self.state.find('X')
        newpos = pos
        if pos is None:
            return None

        if (direction == "N"):
            newpos = pos-3
            if (newpos < 0):
                return None
        elif (direction == "S"):
```

```
            newpos = pos+3
            if (newpos > (self.size*self.size)-1):
                return None
        elif (direction == "O"):
            if (pos % 3 == 0):
                return None
            newpos = pos-1
        elif (direction == "E"):
            if (pos % 3 == 2):
                return None
            newpos = pos+1

        res = Taquin(self)
        tmp = res.state[newpos]
        res.state = res.state.replace('X', 'Y')
        res.state = res.state.replace(self.state[newpos], 'X')
        res.state = res.state.replace('Y', tmp)
        res.history += direction
        return res

    def getChild(self):
        children = []

        if self.move("N") != None:
            children.append(self.move("N"))

        if self.move("S") != None:
            children.append(self.move("S"))

        if self.move("E") != None:
            children.append(self.move("E"))

        if self.move("O") != None:
            children.append(self.move("O"))

        return children

    def dist_manhattan(self, e, state_final):
        # renvoie le nombre de deplacement a faire pour obtenir l'etat final
        pos = self.state.find(e)

        # cherche la position de la tuile dans l'etat final
        pos_final = state_final.state.index(e)
        # deplacements a faire pour obtenir la position finale de la tuile
        h = abs(pos % self.size - pos_final % self.size) + \
            abs(pos // self.size - pos_final // self.size)
        return h

    def get_h_score(self, mod, final_state):
        res = 0
        a = 0
        for e in self.state:
            if (e == "X"):
                continue

```

```
            a += POIDS[mod-1][int(e)] * (self.dist_manhattan(e, final_state))
            res = a // COEFF[mod-1]
        return res

    def get_traveledPath(self):
        return len(self.history)

    def printTaquin(self, deplacement):
        temp = self
        list = []
        for i in deplacement:
            a = temp.move(i)
            if a:
                temp = a
                list.append(a)
        return list


class Algo:

    def __init__(self):
        self.explored_count = 0  # Initialisation du compteur

    def find(self, mode, initial_state, final_state):
        border_set = []  # liste ordonnee de l'ensemble fontiere
        border_dict = {}  # dictionnaire de l'ensemble frontiere
        explored_dict = {}  # savoir si l'etat a deja ete parcouru
        cpt = 0
        # au debut mettre l'etat initial dans frontiere + explored
        initial_state.f = initial_state.get_h_score(mode, final_state)
        border_set.append(initial_state)
        border_dict[initial_state] = initial_state

        while border_set:
            current = border_set.pop(0)

            # supprime de border_dict et ajoute a explored_dict
            del border_dict[current]
            explored_dict[current] = current
            self.explored_count += 1

            # Verifier si l'etat courant de la liste correspond a l'etat final
            if current.state == final_state.state:
                # Si oui, terminer la boucle
                return current

            # recuperer les enfants de ce premier etat de la liste fontiere
            neighbors = current.getChild()

            for neighbor in neighbors:
                # calculer les heuristiques des enfants
                neighbor.f = neighbor.get_traveledPath() + neighbor.get_h_score(mode, final_state)

```

```
                # rechercher dans l ensemble si les etats des enfants ne sont pas deja presents => Si l etat est deja present et a un chemin plus long que le nouveau, on le supprime de l'ensemble et du dictionnaire frontiere
                if neighbor in border_dict:
                    found = border_dict[neighbor]
                    if (found.get_traveledPath() > neighbor.get_traveledPath()):
                        border_set.remove(found)
                        del border_dict[found]
                    else:
                        continue

                # Si l etat est deja present et a un chemin plus court que le nouveau, on oublie le nouveau
                # sinon on l insere dans l ensemble fontiere par ordre d heuristique croissante, et on le reference dans le dictionnaire fontiere
                if neighbor in explored_dict:
                    found = explored_dict[neighbor]

                    if (found.get_traveledPath() > neighbor.get_traveledPath()):
                        explored_dict[neighbor] = neighbor
                    else:
                        continue

                # Ajouter le voisin a la frontiere en triant par ordre croissan
                border_set.append(neighbor)
                border_set.sort(key=lambda x: x.f)
                border_dict[neighbor] = neighbor

        # Si on n a pas trouve d etat final, on renvoie None
        return None

    def get_explored_states_count(self):
        return self.explored_count


class Test:

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

    def getData(self):
        return self.data

    def isSolvable(self):
        # Convertir la chaîne de caractères en une liste d'entiers
        nums = [int(x) for x in self.data if x != 'X']

        # Compter le nombre d'inversions dans la liste
        inversions = 0
        for i in range(len(nums)):
            for j in range(i+1, len(nums)):
                if nums[i] > nums[j]:
```

```
                    inversions += 1

        # Si le nombre d'inversions est pair, le puzzle est soluble
        return inversions % 2 == 0

    def melanger_chaine(self):
        """Mélange une chaîne de caractères"""
        list_car = list(self.data)
        random.shuffle(list_car)
        chaine_melangee = ''.join(list_car)
        return chaine_melangee


# sum_dep = 0
# sum_time = 0
# cpt_ok = 0
# sum_expl = 0
# cpt_bad = 0
# test = Test("01234567X")
# final = Taquin("01234567X")
# for i in range(1000):

#     t = test.melanger_chaine()
#     new_test = Test(t)
#     print("Test", i)
#     if new_test.isSolvable():
#         taquin = Taquin(new_test.getData())  # [0, 1, 2, 3, 'x', 4, 5, 6, 7]

#         monAlgo = Algo()
#         start = time.time()
#         res = monAlgo.find(3, taquin, final)
#         end = time.time()
#         elapsed = end - start
#         print("temps d'execution : ", round(elapsed, 5))
#         print("nombre de deplacements :", len(res.history))
#         print("deplacements realises :", res.history)
#         print("nombre d'états explorés : ",
#               monAlgo.get_explored_states_count())
#         sum_expl += monAlgo.get_explored_states_count()
#         sum_time += round(elapsed, 5)
#         sum_dep += len(res.history)
#         cpt_ok += 1

#     else:
#         print("Pas de solution")
#         cpt_bad += 1
#     print("\n")

# print("Nombre d'états ok : ", cpt_ok)
# print("Nombre d'états initiaux mauvais : ", cpt_bad, "\n")

# print("Nombre d'états explorés : ", round(sum_expl/cpt_ok))
# print("Temps moyen :", round(sum_time/cpt_ok, 5))
# print("Déplacement moyen :", round(sum_dep/cpt_ok))

taquin = Taquin("X76543210")  # [0, 1, 2, 3, 'x', 4, 5, 6, 7]
```

```
final = Taquin("01234567X")


monAlgo = Algo()
start = time.time()
res = monAlgo.find(1, taquin, final)
end = time.time()
elapsed = end - start

print("\n")
cpt = 0

print("Etat initial :")
print("----------------------")
print(taquin)
for i in taquin.printTaquin(res.history):
    cpt += 1
    print("                      ")
    print("Etape : ", cpt)
    print("----------------------")
    print(i)
print("nombre d'états explorés : ", monAlgo.get_explored_states_count())
print("temps d'exécution : ", round(elapsed, 5))
print("nombre de déplacements :", len(res.history))
print("déplacements réalisés :", res.history)
```

projet2.py