Université d'Evry (2022/2023), **IA**, 1–24

MINI PROJET IA

L3 Informatique (CILS)

Camille Berthaud - 20202238 | Issam Azlouk - 20203045

Projet encadré par Jean-Christophe Janodet

Introduction

Abstract

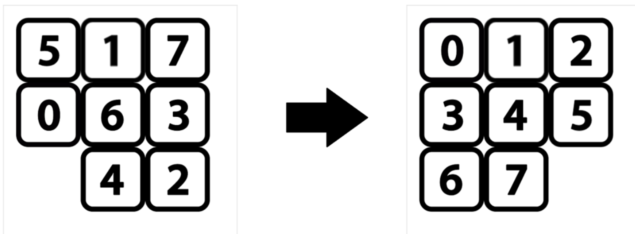
Dans le cadre de notre projet en Intelligence Artificielle, notre objectif était de développer un programme capable de résoudre de manière autonome le célèbre casse-tête du taquin. Pour relever ce défi, il était impératif d'utiliser un algorithme de recherche optimale efficace. Nous avons donc opté pour l'algorithme A* qui, associé à des heuristiques, permet de trouver la solution optimale en réduisant considérablement le nombre de nœuds à explorer. Les heuristiques sont des méthodes qui fournissent des solutions réalisables pour des problèmes complexes.

Sommaire :

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 la case blanche 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.



Le jeu de taquin implique le déplacement des tuiles en les faisant glisser dans la case vide. Cependant, ce déplacement ne peut se faire que pour une seule tuile à la fois. Dans le cadre de notre projet, nous cherchons à déterminer la séquence minimale de déplacements nécessaire pour résoudre le taquin. Pour ce faire, ous avons opté pour l'utilisation de l'algorithme A-star (ou A*).

1.2 L'Algorithme A*

L'algorithme A* est une méthode de recherche de chemin qui permet de trouver le chemin le plus court entre un nœud de départ et un nœud d'arrivée dans un graphe pondéré. Grâce à une approche heuristique, cet algorithme améliore l'efficacité de la recherche en explorant en priorité les nœuds les plus prometteurs. De cette manière, il est possible de trouver rapidement une solution optimale tout en minimisant le nombre de nœuds explorés. 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 dans l'algorithme A* pour classer les nœuds à explorer en priorité. Elle permet de hiérarchiser les nœuds à explorer en fonction de leur potentiel pour atteindre le nœud final avec le coût le plus faible. En effet, les nœuds ayant les plus petites valeurs de f(n) sont explorés en premier. Cette approche permet de rechercher en priorité les nœuds les plus prometteurs, c'est-à-dire ceux qui ont le coût le plus faible et donc le plus de chances d'aboutir à une solution optimale.

Si la fonction heuristique utilisée est admissible, c'est-à-dire toujours inférieure ou égale au coût réel pour atteindre le nœud final, l'algorithme A* est garanti de trouver le chemin le plus court. Cependant, il peut être coûteux en temps de calcul si le graphe à explorer est très grand et complexe. De même, si la fonction heuristique est mal choisie, l'algorithme risque de perdre en efficacité.

2. Explication du code

Le projet se compose de 4 fichiers, chacun correspondant à une classe distincte : Taquin, Algo et Test pour les 3 premiers fichiers, et l'interface graphique pour le 4ème. Cette organisation permet de clarifier le code et facilite la réutilisation de la classe Algo pour la résolution d'autres problèmes complexes.

2.1 Class Taquin

constructeur : Définition d'un état.

Le constructeur de la classe Taquin a pour paramètre une variable pouvant prendre 2 types :

- Une chaîne de caractères représentant un état du taquin.
  Dans ce cas, le constructeur vérifie la longueur de la chaîne. Si la condition est respectée, l'historique des mouvements effectués est stocké et la taille de la grille ainsi que la distance f jusqu'à l'état final sont calculés.
- 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.

__hash()__: Fonction de hachage.
La table de hachage permet de stocker et d'accéder rapidement à des états de Taquin, évitant de parcourir inutilement plusieurs fois les mêmes états. Elle permet d'optimiser les performances de l'algorithme de résolution du problème, en réduisant la complexité temporelle de certaines opérations de recherche dans l'ensemble exploré.

move() et getChild() : Obtention des enfants d'un état.
Pour résoudre le problème, il est essentiel de définir les états et les actions associés. 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.

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. Avec pour coefficient de normalisation respectifs :

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

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.

2.2 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.

bfs() : Algorithme BFS

Cette classe implémente l'algorithme BFS. La méthode bfs() prend en entrée l'état initial et l'état final. Cette méthode retourne l'état final ainsi que le chemin parcouru pour l'atteindre.

La méthode bfs() initialise une file d'attente queue pour stocker les états à explorer et un dictionnaire explored_dict pour enregistrer les états déjà explorés. L'état initial est ajouté à la file d'attente et marqué comme exploré dans le dictionnaire.

Ensuite, l'algorithme parcourt une boucle, tant que la file d'attente n'est pas vide. L'algorithme extrait le premier élément de la file d'attente et le marque comme exploré. Si cet état correspond à l'état final recherché, alors la boucle est terminée et l'état final est renvoyé.

Si l'état courant n'est pas l'état final, l'algorithme récupère les enfants (voisins) de l'état courant et les ajoute à la file d'attente s'ils n'ont pas déjà été explorés. Le processus continue jusqu'à ce que tous les états accessibles à partir de l'état initial soient explorés ou que l'état final soit atteint.

Si aucun chemin menant à l'état final n'est trouvé, la méthode renvoie None.

2.3 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() : Vérifie 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.

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

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 coups : le nombre d'états faisant partie de 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 observer que chaque heuristique permet d'arriver à la solution avec un nombre de coups différent. Nous pouvons également constater 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 :

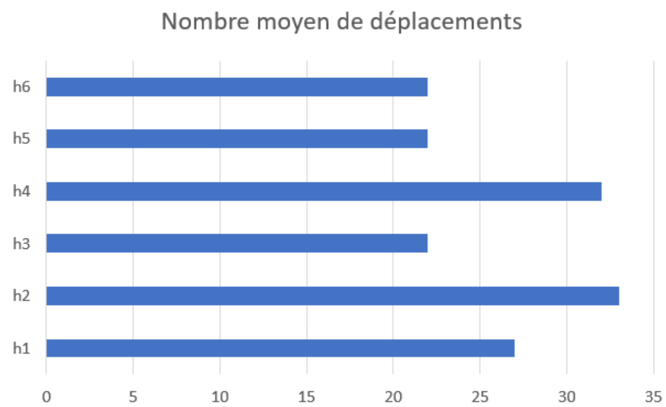
- Le nombre de coups est optimal (le plus bas).
- Le temps de résolution et le nombre de noeuds explorés sont dans la moyenne.

Les heuristiques h3 et h5 permettent également d'avoir une solution 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 :

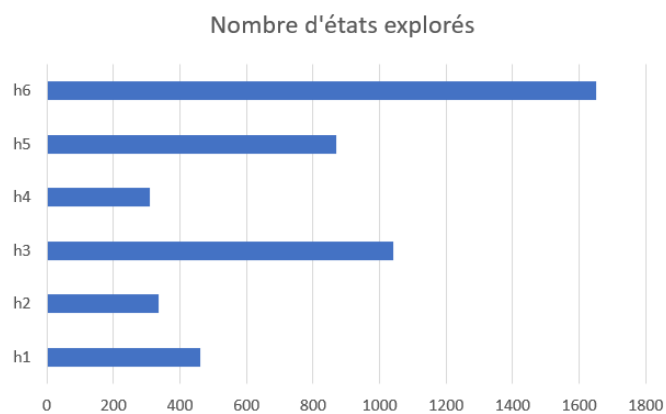
Nous allons maintenant vérifier si le test précédent est représentatif, en testant sur un échantillon de 1000 taquins générés aléatoirement. Les résultats obtenus sont les moyennes des différents tests. Dans les tests réalisés en moyenne 50 % des taquins sont résolvables.

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.

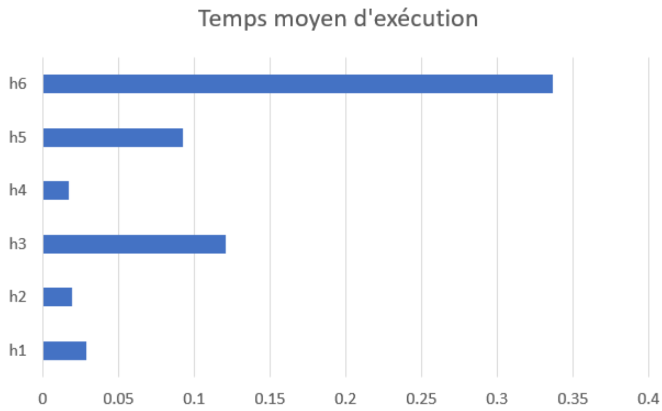
Le nombre d'états explorés :



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 déplacements.

Le temps d'exécution :



Nous pouvons constater que les heuristiques qui expandent le plus de noeuds sont les moins rapides. En revanche plus 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 plus faible en moyenne.

3.3 Test de performances entre les différents algorithmes

Dans le but d'étudier l'efficacité de chaque algorithme, nous allons générer 1000 taquins aléatoires et comparer leurs performances.

A noter que les valeurs de A* correspondent aux valeurs de h5, vu précédemment.

Voici les résultats :

	Temps de résolution	Nombre de noeuds explorés	Nombre de coups
A*	0.09275s	870	22
BFS	1.52298s	90270	22

Nous pouvons observer que l'utilisation d'un algorithme A* est plus adapté qu'un algorithme utilisant une recherche aveugle (sans heuristiques).
Cette différence de résultat est expliquée par :

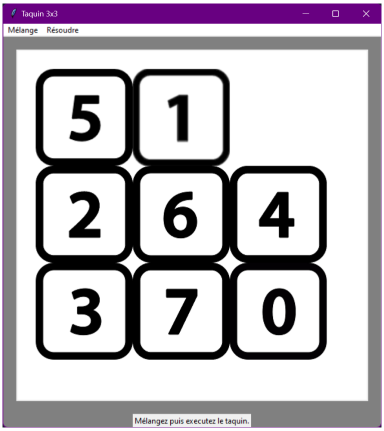
- BFS qui explore tous les états proches d'un sommet **s** avant de continuer, cependant il trouve toujours le chemin le plus court. Cet algorithme a donc une complexité en espace mémoire importante.

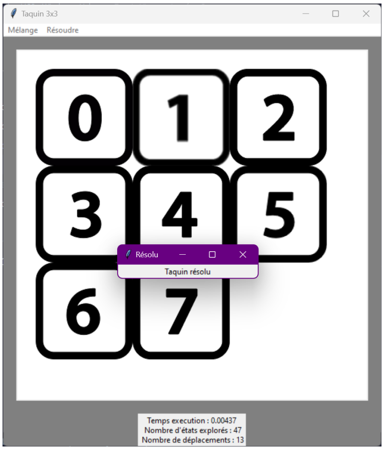
A* est plus efficace que BFS dans la résolution du problème du taquin car il utilise une fonction d'évaluation pour guider la recherche vers la solution optimale.

4. Interface graphique

Nous avons développé une interface graphique qui permet d'afficher les déplacements du taquin en temps réel. Cette interface offre une expérience de jeu plus immersive en permettant aux joueurs de voir les mouvements du taquin se dérouler sous leurs yeux.

Visualisations de l'interface :





5. Extensions possibles

Bien que le jeu de taquin classique soit joué sur une grille de 3×3, il existe des versions étendues du jeu qui permettent aux joueurs de résoudre des puzzles plus complexes en utilisant des grilles de tailles variables. Il aurait été intéressant de tester le jeu de taquin pour une grille de dimension $n \times n$.

6. 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.

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.

7. Annexes

```
import math

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):

        #Si paramètre chaine de caractère
        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 = ''

        #Si Paramètre taquin
        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):
        return ''.join(str(self.state))

    #  Fonction de déplacement
    def move(self, direction):

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

        if direction == "N":
            new_position = pos - self.size
            if (new_position < 0):
                return None
```

```python

        elif direction == "S":
            new_position = pos + self.size
            if new_position > (self.size * self.size) - 1:
                return None

        elif direction == "O":
            if pos % self.size == 0:
                return None
            new_position = pos - 1

        elif direction == "E":
            if pos % self.size == self.size - 1:
                return None
            new_position = pos + 1

        else:
            return None

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

    # Fonction retournant tout les enfants d'un état
    def getChild(self):
        children = []

        if self.move("N") is not None:
            children.append(self.move("N"))

        if self.move("S") is not None:
            children.append(self.move("S"))

        if self.move("E") is not None:
            children.append(self.move("E"))

        if self.move("O") is not None:
            children.append(self.move("O"))

        return children

    # Fonction de test de distance manhattan
    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

    # Donne le coût de l'heuristique
    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

    # Renvoie le nombres de déplacement pour arriver à l'état final
    def get_traveledPath(self):
        return len(self.history)

    # Fonction d'affichage du taquin
    def printTaquin(self, deplacement):
        temp = self
        list_t = []
        for i in deplacement:
            a = temp.move(i)
            if a:
                temp = a
                list_t.append(a)
        return list_t
```

Taquin.py

```
class Algo:

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

    # Récupère le nombre d'etats exploré
    def get_explored_states_count(self):
        return self.explored_count

    # Fonction d'algorithme de recherche A*
    def a_star(self, mode, initial_state, final_state):
        border_set = []  # Liste ordonnée de l'ensemble frontière
        border_dict = {}  # Dictionnaire de l'ensemble frontière
        explored_dict = {}  # Dictionnaire pour vérifier si un état a déjà été exploré

        # Insérer l'état initial dans la frontière et le dictionnaire des états explorés
        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:
            # Sélectionner et supprimer le premier élément de la frontière
```

```
        current = border_set.pop(0)
        del border_dict[current]
        explored_dict[current] = current
        self.explored_count += 1

        # Vérifier si l'état courant correspond à l'état final
        if current.state == final_state.state:
            return current  # Si oui, terminer la boucle et renvoyer l'état courant

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

        for child in children:
            # Calculer l'heuristique (score) de l'enfant
            child.f = child.get_traveledPath() + child.get_h_score(mode, final_state)

           # rechercher dans l'ensemble si les etats des enfants ne sont pas deja presents =

            if child in border_dict:
                found = border_dict[child]
                # 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 (found.get_traveledPath() > child.get_traveledPath()):
                    border_set.remove(found)
                    del border_dict[found]
                else:
                    # Si l'etat est deja present et l'ancien a un chemin plus court que le nouveau, on oublie le
                    # nouveau
                    continue

            # on insere le nouvel élément dans l ensemble fontiere par ordre d heuristique croissante,
            # et on le reference dans le dictionnaire fontiere
            if child in explored_dict:
                found = explored_dict[child]

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

            # Insérer le voisin dans la frontière en triant par ordre croissant d'heuristique
            border_set.append(child)
            border_set.sort(key=lambda x: x.f)
            border_dict[child] = child
            explored_dict[child] = child

            # Si aucun chemin menant à l'état final n'est trouvé,
```

```
renvoyer None
        return None

    # Fonction d'algorithme de recherche en largeur
    def bfs(self, initial_state, final_state):
        queue = []  # File d'attente pour les états à explorer
        explored_dict = {}  # Dictionnaire pour vérifier si un état a déjà été exploré

        # Insérer l'état initial dans la file d'attente et le dictionnaire des états explorés
        queue.append(initial_state)
        explored_dict[initial_state] = initial_state

        while queue:
            # Sélectionner et supprimer le premier élément de la file d'attente
            current = queue.pop(0)
            self.explored_count += 1

            # Vérifier si l'état courant correspond à l'état final
            if current.state == final_state.state:
                return current  # Si oui, terminer la boucle et renvoyer l'état courant

            # Récupérer les enfants de l'état courant
            children = current.getChild()

            for neighbor in children:
                # Vérifier si le voisin a déjà été exploré
                if neighbor in explored_dict:
                    continue

                # Ajouter le voisin à la file d'attente et le marquer comme exploré
                queue.append(neighbor)
                explored_dict[neighbor] = neighbor

        # Si aucun chemin menant à l'état final n'est trouvé, renvoyer None
        return None
```

Algo.py

```
from Taquin import Taquin
from Algo import Algo
import random
import time


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

    def isSolvable(self):
        # Convertir la chaîne de caractères en une liste d'entiers hormis le "X"
```

```
        nums = [int(x) for x in self.data if x != 'X']

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

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

    # Fonction qui renvoie un taquin aléatoire
    def shuffle(self):
        # Mélange une chaine de caractères
        list_car = list(self.data)
        random.shuffle(list_car)
        return [''.join(list_car), list_car]  # Reconcatene la chaine

    # Fonction de test
    def test(self, taquin_t, taquin_finale, choiceAlgo):

        if taquin_t.isSolvable():
            taquin = Taquin(taquin_t.data)
            algo = Algo()

            start = time.time()
            if choiceAlgo == "a_star":
                res = algo.a_star(5, taquin, taquin_finale)
            elif choiceAlgo == "bfs":
                res = algo.bfs(taquin, taquin_finale)
            end = time.time()
            elapsed = end - start

            explored = algo.get_explored_states_count()
            print("Temps d'exécution : ", round(elapsed, 5))
            print("Nombre de déplacements :", len(res.history))
            print("Déplacements realises :", res.history)
            print("Nombre d'états explorés : ", explored)
            return [taquin_t, res, elapsed, explored]
        else:
            print("Taquin non résolvable.")
            return [taquin_t, 0]

    # Fonction permettant de faire 1000 tests
    def TestForThousand(self, taquin_finale, choiceAlgo):
        sum_dep = 0  # Somme des nombres de déplacements
        sum_time = 0  # Somme des temps d'exécutions
        sum_expl = 0  # Somme d'états explorés
        cpt_ok = 0  # Compteur de configurations soluble
        cpt_bad = 0  # Compteur de configurations non soluble

        for i in range(1000):
            print("Test :", i)

            taquin_rdm = Test(self.shuffle()[0])
            test = self.test(taquin_rdm, taquin_finale, choiceAlgo)
```

```python

            if test[0].isSolvable():
                sum_expl += test[3]
                sum_time += round(test[2], 5)
                sum_dep += len(test[1].history)
                cpt_ok += 1

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

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

        if cpt_ok > 0:
            print("Nombre d'états moyen explorés : ", round(sum_expl / cpt_ok))
            print("Temps moyen :", round(sum_time / cpt_ok, 5))
            print("Déplacement moyen :", round(sum_dep / cpt_ok))

        return [cpt_ok, cpt_bad, sum_expl, sum_time, sum_dep]
```

Test.py

```python
import tkinter
import tkinter as tk
from tkinter import TOP, NW
from Taquin import Taquin
from Test import Test

# Définition d'une liste des valeurs possibles pour le jeu de taquin
valeur = [0, 1, 2, 3, 4, 5, 6, 7, 8]
global label

# Fonction qui convertit une liste de chaînes de caractères en une liste de valeurs numériques ou de la chaîne 'X'
def changeTypeValeur(taquin):
    global valeur
    valeur = [int(x) if x != "X" else 8 for x in taquin]
    return valeur


# Fonction qui convertit une liste de valeurs numériques ou la chaîne 'X' en une liste de chaînes de caractères
def changeTypeTaquin(valeur):
    n_val = [str(x) if x != 8 else 'X' for x in valeur]
    return n_val


# Fonction qui mélange le jeu de taquin en utilisant la classe Test
def melanger_taquin():
    global valeur
    taquin = Test("01234567X").shuffle()[1]
    valeur = changeTypeValeur(taquin)
    print("Nouveau taquin : ", str(valeur))

```

```python

# Fonction qui met à jour l'interface graphique pour afficher le jeu de taquin actuel
def update_taquin(canvas, photos):
    canvas.delete("all")
    for k in range(len(photos)):
        canvas.create_image((30 + 150 * (k % 3)), 30 + (150 * (k // 3)), anchor=NW, image=photos[valeur[k]])


# Fonction qui résout le jeu de taquin en utilisant la classe Test et affiche les résultats dans une nouvelle fenêtre
def resoudre_taquin(canvas, photos, new_fen, ChoixAlgo):
    global valeur
    n_valeur = changeTypeTaquin(valeur)
    n_list = ''.join(n_valeur)
    test = Test(n_list)
    taquin = Taquin(n_list)
    nt = Test("01234567X").test(test, Taquin("01234567X"), ChoixAlgo)

    if nt[1] == 0:
        fenetre_err = tk.Tk()
        tk.Label(fenetre_err, text="Taquin non resolvable en A* \n Veuillez mélanger de nouveau", width=50).pack()
    else:
        a = nt[1].history
        cpt = 0
        print("Etat initial :", taquin)

        n_lab =tkinter.Label(new_fen, text="Execution du taquin étape : " + str(cpt))
        n_lab.pack()

        # Affichage des étapes
        for i in taquin.printTaquin(a):
            cpt += 1
            v = list(str(i))
            # Affichage des étapes
            print(f"Etape {cpt} : Nouvel Etat : {v} ")
            n_lab.config(text="Execution du taquin étape : " + str(cpt))

            # Mise à jour
            valeur = changeTypeValeur(v)
            update_taquin(canvas, photos)
            new_fen.after(500)
            new_fen.update()

        # Affichage des résultats
        n_lab.config(text="Temps execution : " + str(round(nt[2], 5)) + "\n Nombre d'états explorés : " + str(nt[3]) + "\n Nombre de déplacements : " + str(len(a)))
        # Affichage de la fenêtre résolue
        t_resolve = tk.Tk()

        # Taille de l'ecran
        largeur_ecran = t_resolve.winfo_screenwidth()
        hauteur_ecran = t_resolve.winfo_screenheight()
```

```python

        # Calculer les coordonnées pour centrer la fenêtre
        x = (largeur_ecran - t_resolve.winfo_reqwidth()) / 2
        y = (hauteur_ecran - t_resolve.winfo_reqheight()) / 2

        # Définition de la position de la fenêtre
        t_resolve.geometry("+%d+%d" % (x - 100, y))

        # Affichage
        t_resolve.title('Résolu')
        tk.Label(t_resolve, text="Taquin résolu", width=30).pack()
        print("Taquin Résolu.")
        t_resolve.protocol("WM_DELETE_WINDOW", lambda: [n_lab.destroy(), t_resolve.quit(), t_resolve.destroy()])
        t_resolve.mainloop()


# Affichage de la fenêtre du taquin
def new_window():
    # Destruction de l'ancienne fenêtre
    maFenetre.destroy()
    # Créer une nouvelle fenêtre
    nouvelle_fen = tk.Tk()
    nouvelle_fen.title('Taquin 3x3')
    nouvelle_fen.configure(background='grey')

    # Taille de l'ecran
    largeur_ecran = nouvelle_fen.winfo_screenwidth()
    hauteur_ecran = nouvelle_fen.winfo_screenheight()

    # Calculer les coordonnées pour centrer la fenêtre
    x = (largeur_ecran - nouvelle_fen.winfo_reqwidth()) / 2
    y = (hauteur_ecran - nouvelle_fen.winfo_reqheight()) / 2

    # Définition de la position de la fenêtre
    nouvelle_fen.geometry("+%d+%d" % (x- 200, y- 250))

    # Création de l'environnement du taquin
    canvas = tk.Canvas(nouvelle_fen, width=540, height=180 * 3, bg='white')
    canvas.pack(side=TOP, padx=20, pady=20)

    # Chargement des images
    photos = [tk.PhotoImage(file=f"./number_buttons/{i}.png") for i in range(0,9)]

    # Affichage initial du taquin
    update_taquin(canvas, photos)

    n_label = tkinter.Label(nouvelle_fen, text="Mélangez puis executez le taquin.")
    n_label.pack()
    n_label2 = tkinter.Label(nouvelle_fen, text="Mélangez puis executez le taquin.")

    # Création de la barre de menu
    menubar = tk.Menu(nouvelle_fen)
```

```python

    # Menu "Mélanger / Quitter"
    menu1 = tk.Menu(menubar, tearoff=0)
    menu1.add_command(label="Mélanger", command=lambda: [melanger_taquin(), update_taquin(canvas, photos)])
    menu1.add_separator()
    menu1.add_command(label="Quitter", command=nouvelle_fen.destroy)
    menubar.add_cascade(label="Mélange", menu=menu1)

    # Menu "Résoudre"
    menu2 = tk.Menu(menubar, tearoff=0)
    menu2.add_command(label="A*", command=lambda: [n_label.pack_forget(), resoudre_taquin(canvas, photos, nouvelle_fen, "a_star"), n_label.pack()])
    menu2.add_command(label="Recherche en longueur", command=lambda: [n_label.pack_forget(), resoudre_taquin(canvas, photos, nouvelle_fen, "dfs"), n_label.pack()])
    menu2.add_command(label="Recherche en largeur", command=lambda: [n_label.pack_forget(), resoudre_taquin(canvas, photos, nouvelle_fen, "bfs"), n_label.pack()])
    menubar.add_cascade(label="Résoudre", menu=menu2)

    nouvelle_fen.config(menu=menubar)

    nouvelle_fen.mainloop()


# Création de la fenêtre principale (menu de choix de taquin)
maFenetre = tk.Tk()
maFenetre.title('Jeu du Taquin')
maFenetre.configure(background="grey")

# Taille de l'ecran
largeur_ecran = maFenetre.winfo_screenwidth()
hauteur_ecran = maFenetre.winfo_screenheight()

# Calculer les coordonnées pour centrer la fenêtre
x = (largeur_ecran - maFenetre.winfo_reqwidth()) / 2
y = (hauteur_ecran - maFenetre.winfo_reqheight()) / 2

# Définition de la position de la fenêtre
maFenetre.geometry("+%d+%d" % (x-100, y))

# Texte de la fenêtre principale
tk.Label(maFenetre, text="Choix du taquin", width=50).pack()

# Création du bouton "Lancer Taquin 3x3"
button = tk.Button(maFenetre, text="Lancer Taquin 3x3", width=50, command=new_window)

button.pack()
# Laisser la fenêtre principale active
maFenetre.mainloop()
```

Interface.py