Université d'Évry Val d'Essonne
Double Licence Droit et Informatique

Exploration et amélioration des méthodes de résolution pour le problème du Taquin

Auteur :
Teoman SOYKAN

Date :
04 Avril 2023

Rapport de projet

Année académique :
2022-2023

1 Introduction

Le taquin est un casse-tête qui consiste en un plateau carré composé de tuiles numérotées et d'un trou. L'objectif du jeu est de déplacer les tuiles pour atteindre un état final donné en un nombre limité de mouvements. Dans ce projet, nous étudions l'implémentation de l'algorithme A* pour résoudre le problème du taquin 3x3, en utilisant différentes heuristiques. Nous comparons les performances de ces heuristiques en fonction du nombre d'états explorés, du temps de calcul et de la longueur de la solution obtenue.

2 Problématique et objectifs

Le problème du taquin peut être formulé comme suit : étant donné un état initial et un état final, trouver la séquence de mouvements qui permet de passer de l'état initial à l'état final en un minimum de mouvements. L'objectif de ce projet est de mettre en œuvre l'algorithme A* pour résoudre ce problème et d'étudier les performances des différentes heuristiques utilisées.

3 Méthodologie

Dans cette section, nous détaillons l'implémentation de l'algorithme A* et les heuristiques utilisées pour résoudre le problème du taquin 3x3.

3.1 Algorithme A*

L'algorithme A* est un algorithme de recherche informée qui explore les états en fonction de leur coût total estimé, qui est la somme du coût réel pour atteindre un état donné depuis l'état initial (noté **g(E)**) et du coût estimé pour atteindre l'état final à partir de cet état (noté **h(E)**). La fonction d'évaluation de l'algorithme A* est définie comme suit : **f(E) = g(E) + h(E).**

L'implémentation de l'algorithme A* nécessite de définir les structures de données pour représenter les états, l'ensemble frontière et l'ensemble exploré. Pour les états, nous utilisons une structure de données qui stocke la configuration du plateau, le chemin le plus court pour atteindre l'état depuis l'état initial et sa longueur. Pour l'ensemble frontière, nous utilisons une file de priorité d'états, ordonnée par valeur croissante de **f(E)**. Enfin, pour l'ensemble exploré, nous utilisons un arbre binaire de recherche ou une table de hachage pour stocker les états visités.

La fonction d'évaluation est une combinaison de la distance parcourue depuis l'état initial **(coût g)** et d'une estimation du coût restant pour atteindre l'état final **(coût h)**, appelée heuristique. L'algorithme explore les états en priorisant ceux qui ont le plus petit coût total **(g + h)**.

Dans le cadre du problème du taquin, plusieurs heuristiques peuvent être utilisées pour estimer le coût restant. Le programme implémenté utilise six heuristiques différentes, basées sur des pondérations et des distances de Manhattan. Les heuristiques permettent d'orienter la recherche de l'algorithme A* vers les états les plus prometteurs, réduisant ainsi le nombre d'états explorés et le temps de résolution.

La complexité temporelle et spatiale de l'algorithme A* dépend de l'heuristique utilisée. En général, une heuristique plus précise réduit la complexité temporelle, car moins d'états sont explorés. La complexité spatiale est également influencée par l'heuristique, car elle détermine le nombre d'états stockés dans la mémoire pendant la recherche.

3.2 Implémentation de l'agorithme A*

1. **Initialisation** : Créez une instance de la classe `Taquin` avec l'état initial et initialisez la frontière (la liste des nœuds à explorer) en utilisant une liste de priorités (`heapq`). Ajoutez le nœud initial avec sa valeur de fonction f (calculée en utilisant l'heuristique choisie) à la frontière.
2. **Exploration** : Tant que la frontière n'est pas vide :
   — Retirez le nœud ayant la plus petite valeur f de la frontière (utilisez `heapq.heappop()` pour obtenir et supprimer le nœud de la frontière).
   — Si l'état du nœud actuel correspond à l'état final, reconstruisez et retournez la solution en remontant les étapes depuis le nœud actuel jusqu'au nœud initial.
   — Ajoutez l'état du nœud actuel à l'ensemble des états explorés et incrémente le nombre d'états explorés.

— Générez les voisins du nœud actuel en utilisant la méthode `get_neighbors()` et, pour chaque voisin non exploré, ajoutez-le à la frontière avec sa valeur f (utilisez `heapq.heappush()` pour ajouter le nœud à la frontière).

3. **Aucune solution trouvée** : Si la frontière est vide et qu'aucune solution n'a été trouvée, retournez `None` et le nombre d'états explorés.

L'heuristique utilisée pour guider l'algorithme A* est déterminée par la méthode `h()` de la classe `Taquin`. Selon l'heuristique choisie, cette méthode calcule la somme pondérée des distances de Manhattan entre les tuiles actuelles et leurs positions cibles. Les poids sont définis dans la matrice `weights`, et le facteur de pondération `pk` est choisi en fonction de l'heuristique. La fonction f est ensuite calculée comme la somme de la distance parcourue depuis l'état initial g et de la valeur heuristique h.

3.3 Heuristiques

Nous avons choisi six heuristiques pour résoudre le problème du taquin 3x3. Toutes ces heuristiques sont basées sur la distance de Manhattan pondérée réduite. La distance de Manhattan est une mesure de la distance entre deux points dans une grille en ne se déplaçant qu'horizontalement ou verticalement. La distance de Manhattan pondérée réduite est calculée comme suit :

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

où $\varepsilon_E(i)$ est la distance élémentaire de la tuile i dans l'état E, $\pi_k(i)$ est le poids de la tuile i pour l'heuristique h_k, et ρ_k est un coefficient de normalisation.

Les poids $\pi_k(i)$ et les coefficients de normalisation ρ_k sont définis comme suit pour les différentes heuristiques :

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

TABLE 1 – Poids des tuiles pour les différentes heuristiques

Heuristique	Coefficient de normalisation
h_1	4
h_2, h_3	1
h_4, h_5	4
h_6	1

TABLE 2 – Coefficients de normalisation pour les différentes heuristiques

4 Étude expérimentale

Dans cette étude expérimentale, les performances de 6 heuristiques (h1 à h6) ont été analysées pour résoudre le problème du taquin en utilisant des cas de difficulté variable : facile, moyen et difficile. Par la suite, une analyse a été réalisée sur des séries de 10 et 100 exécutions aléatoires afin d'évaluer leur efficacité globale. Les résultats détaillés pour chaque scénario sont présentés ci-dessous :

4.1 Comparaison des heuristiques pour différents niveaux de difficulté

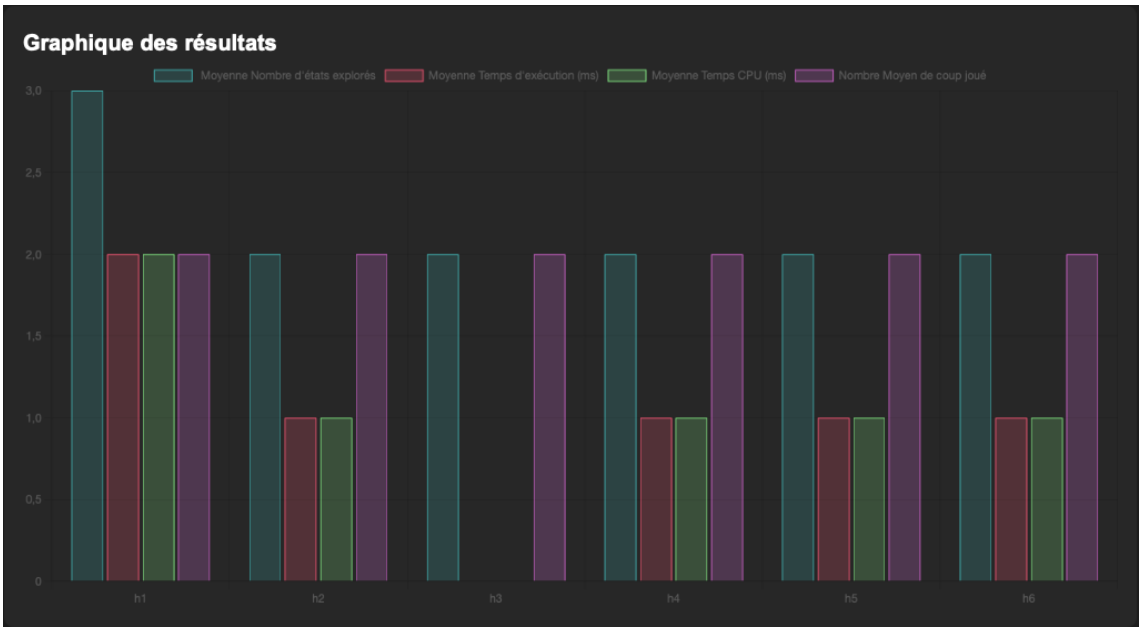
4.1.1 Taquin facile

1	2	3
4	0	6
7	5	8

Dans le cas du table 3, toutes les heuristiques ont résolu le problème avec un faible nombre d'états explorés et un nombre de coups minimal (2). Les temps d'exécution étaient également très similaires, avec h3 étant légèrement plus rapide que les autres.

TABLE 3 – Résultats pour les taquins faciles

Heuristique	États explorés	Temps d'exécution (ms)	Nombre de coups
h1	3	2	2
h2	2	1	2
h3	2	0.5	2
h4	2	1	2
h5	2	1	2
h6	2	1	2



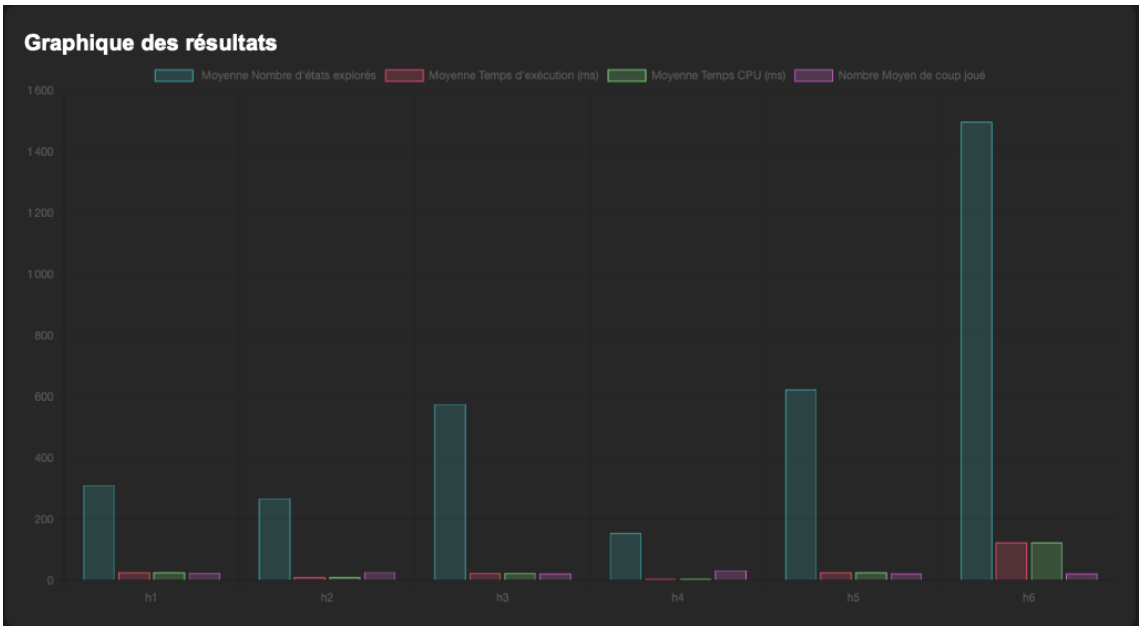
4.1.2 Taquin Moyen

0	1	2
3	4	5
6	7	8

Dans ce scénario du table 4, h3, h5 et h6 ont produit le plus faible nombre de coups (22). Cependant, h6 a présenté un temps d'exécution considérablement plus long (123,5 ms). h4 a généré le plus grand nombre de coups (34), mais avec un temps d'exécution beaucoup plus court (3,67 ms).

TABLE 4 – Résultats pour les taquins moyens

Heuristique	États explorés	Temps d'exécution (ms)	Nombre de coups
h1	311	26	24
h2	268	11	28
h3	576	24	22
h4	155	3.67	34
h5	624	26	22
h6	1498	123.5	22



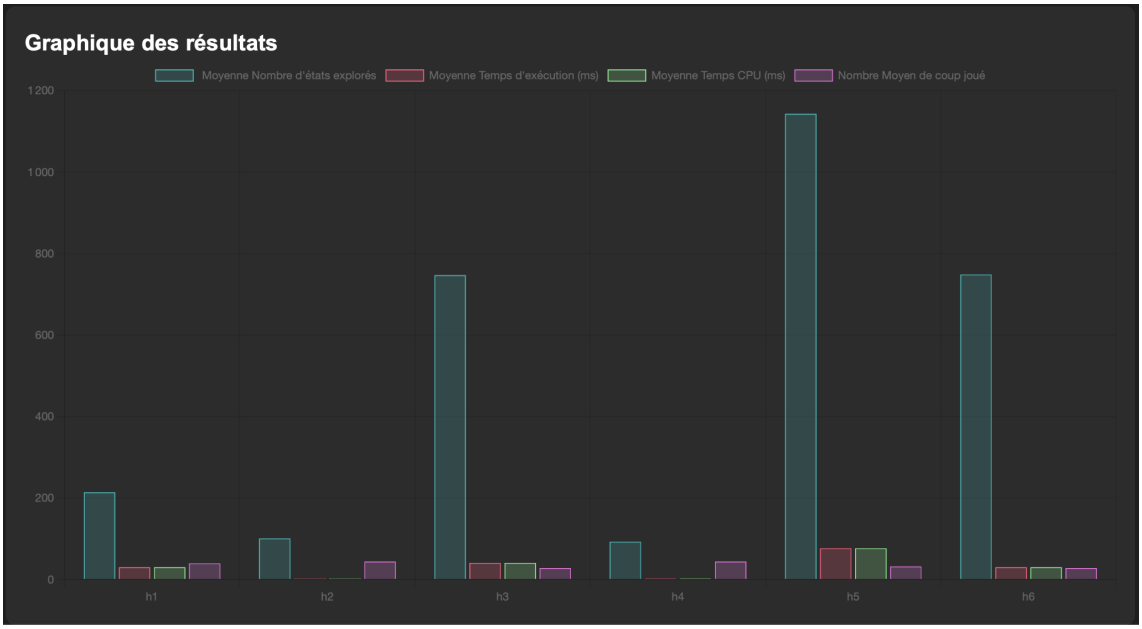
4.1.3 Taquin Difficile

0	8	7
6	5	4
3	2	1

Ici dans la table 5, les heuristiques h3 et h6 ont obtenu les meilleurs résultats en termes de nombre de coups (28), mais h6 a présenté un temps d'exécution plus long (34 ms contre 22 ms pour h3). h1 et h2 ont produit des résultats moins optimaux avec un nombre de coups plus élevé (40 et 44, respectivement).

TABLE 5 – Résultats pour les taquins difficiles

Heuristique	États explorés	Temps d'exécution (ms)	Nombre de coups
h1	214	23	40
h2	101	3	44
h3	747	22	28
h4	93	3.67	44
h5	1143	83	32
h6	749	34	28



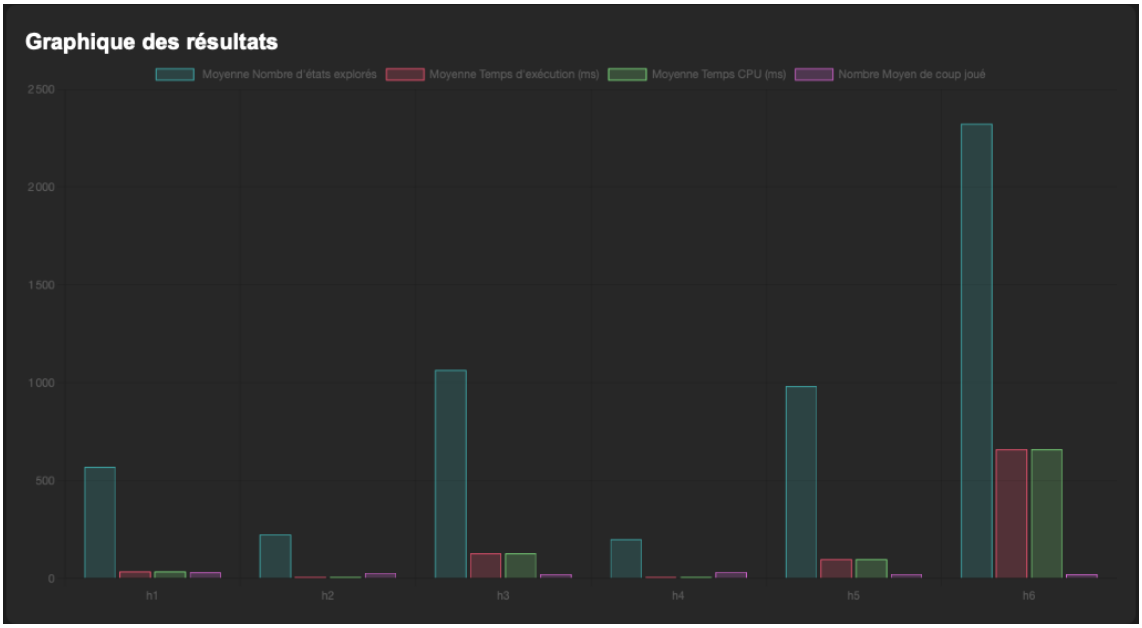
4.2 Comparaison des heuristiques pour 10 et 100 exécutions aléatoires

4.2.1 10 exécutions aléatoires

Pour la Table 6, sur 10 exécutions aléatoires, h6 a obtenu le meilleur résultat en termes de nombre de coups (22,44), mais avec un temps d'exécution beaucoup plus long (661 ms). h3 a également montré de bonnes performances avec un nombre de coups similaire (23,2) et un temps d'exécution plus court (127,7 ms).

TABLE 6 – Résultats pour une moyenne de 10 exécutions aléatoires

Heuristique	États explorés	Temps d'exécution (ms)	Nombre de coups
h1	569.91	35.18	31
h2	224.8	6.4	28.9
h3	1065.6	127.7	23.2
h4	200.36	5.73	33.09
h5	983.5	97.8	23.3
h6	2325	661	22.44

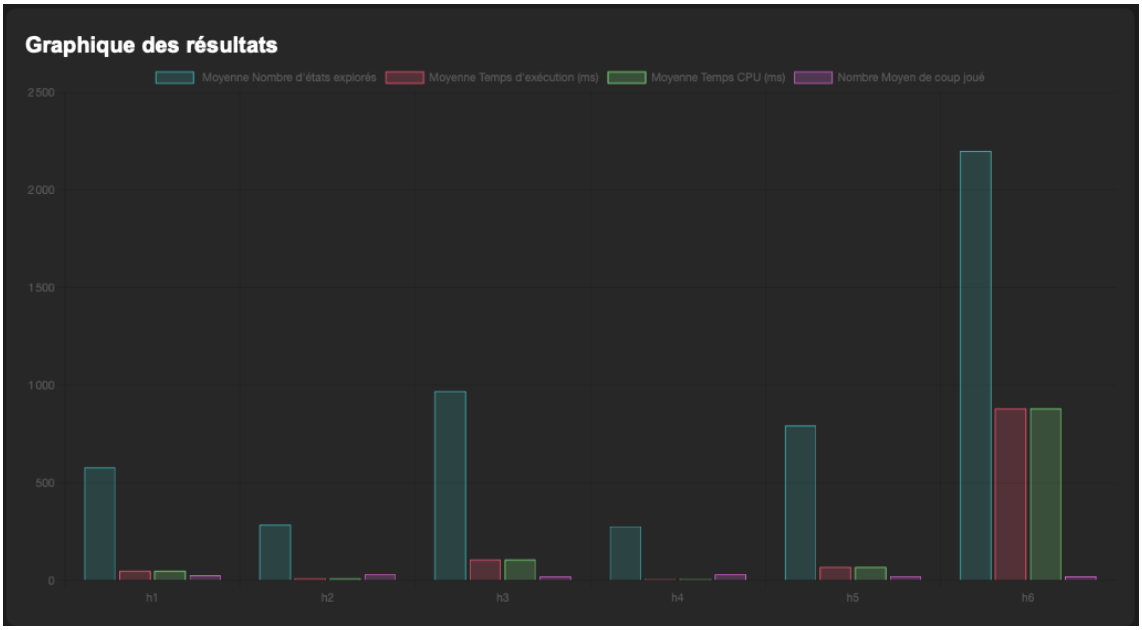


4.2.2 100 exécutions aléatoires

Pour la table 7, dans ce scénario, h6 a de nouveau obtenu les meilleurs résultats en termes de nombre de coups (22,17), mais avec un temps d'exécution très long (881,94 ms). h3 et h5 ont obtenu des résultats similaires en termes de nombre de coups (22,76 et 22,79, respectivement), avec des temps d'exécution beaucoup plus courts (107,3 ms et 68,86 ms, respectivement).

TABLE 7 – Résultats pour une moyenne de 100 exécutions aléatoires

Heuristique	États explorés	Temps d'exécution (ms)	Nombre de coups
h1	579.21	50	26.87
h2	285.74	10.2	32.5
h3	970.45	107.3	22.76
h4	279.15	9.72	32.19
h5	794.47	68.86	22.79
h6	2220.84	881.94	22.17



En conclusion, l'heuristique h3 semble être un choix équilibré pour résoudre le problème du taquin dans divers contextes, offrant une bonne performance en termes de nombre de coups et de temps d'exécution. Cependant, si l'on privilégie le nombre de coups au détriment du temps d'exécution, h6 pourrait être un choix approprié. Il est important de noter que la performance optimale d'une heuristique dépend du contexte et des priorités, telles que la résolution rapide du problème ou l'obtention d'une solution avec un nombre de coups minimal.

4.3 Différences temps d'exécutions et CPU

Le temps d'exécution est mesuré à l'aide de la fonction time.time() avant et après l'exécution de la fonction solve_taquin(). Cette méthode mesure le temps total écoulé entre le début et la fin de l'exécution, y compris le temps d'attente pour les entrées/sorties ou autres tâches non liées à l'exécution du programme.

Le temps CPU, quant à lui, est mesuré à l'aide de la fonction os.times()[0] avant et après l'exécution de la fonction solve_taquin(). Cette méthode mesure le temps processeur utilisé par le programme, c'est-à-dire le temps que le processeur passe à exécuter le programme, sans compter le temps d'attente pour les entrées/sorties ou autres tâches non liées à l'exécution du programme.

Il est normal que ces deux mesures puissent donner des résultats légèrement différents. En effet, le temps CPU mesure uniquement le temps que le processeur a passé à exécuter le programme, sans tenir compte de tout autre facteur externe qui pourrait influencer le temps total d'exécution. À l'inverse, le temps d'exécution mesure le temps total écoulé, y compris les temps d'attente pour les entrées/sorties ou les tâches non liées à l'exécution du programme.

5 Extensions possibles

1. Expérimenter avec d'autres heuristiques, comme la distance de Chebyshev ou la distance de Canberra : L'heuristique utilisée dans ce programme est la distance de Manhattan, qui est une heuristique couramment utilisée pour résoudre le Taquin. Cependant, il existe d'autres heuristiques, telles que la distance de Chebyshev ou la distance de Canberra, qui peuvent être utilisées pour résoudre le Taquin. L'expérimentation avec ces autres heuristiques pourrait aider à déterminer si l'une d'entre elles peut améliorer les performances de l'algorithme A* pour résoudre le Taquin.
2. Implémenter des stratégies de recherche bidirectionnelle pour améliorer l'efficacité de l'algorithme : L'algorithme A* utilisé dans ce programme recherche depuis l'état initial jusqu'à l'état final. Une autre approche consiste à rechercher simultanément depuis l'état initial et l'état final en utilisant deux arbres de recherche. Cette approche peut être plus efficace que la recherche à partir de l'état initial seul, car elle peut réduire l'espace de recherche nécessaire pour trouver une solution. Cela pourrait potentiellement accélérer la recherche de solutions pour les taquins de grande taille.

3. Tester des techniques de prétraitement et de réduction de l'espace de recherche, comme le partitionnement des taquins de grande taille : Pour les taquins de grande taille, l'espace de recherche peut être très grand, ce qui peut rendre la recherche de solutions très difficile. Une approche pour réduire l'espace de recherche consiste à diviser le Taquin en sous-taquins plus petits et à résoudre chaque sous-taquin individuellement. Cette approche peut potentiellement accélérer la recherche de solutions pour les taquins de grande taille.
4. Comparer les performances de l'algorithme A* avec d'autres algorithmes de recherche, tels que IDA* ou BFS : Il existe plusieurs autres algorithmes de recherche qui peuvent être utilisés pour résoudre le Taquin, tels que IDA* ou BFS. La comparaison de ces algorithmes avec l'algorithme A* pourrait aider à déterminer lequel est le plus efficace pour résoudre le Taquin. Cela peut également fournir des informations sur les avantages et les inconvénients de chaque algorithme en termes de complexité temporelle et d'utilisation de la mémoire.

6 Interface Web

Dès le début, une version web du programme a été développée pour faciliter et accélérer la création d'une maquette de notre programme Python. La version web, entièrement fonctionnelle, tire parti du langage JavaScript pour offrir de meilleures performances et une plus grande flexibilité dans la conception de l'interface. De plus, la version web constitue une véritable transposition de notre programme Python, accessible sur tout appareil équipé d'un navigateur. Voici le lien du Site Web : **https ://ihw-ts.github.io/Taquin_Game_IA/**

7 Conclusion générale

En somme, l'algorithme **A*** réussit systématiquement à trouver une solution au problème du Taquin. Toutefois, en fonction des heuristiques utilisées, les solutions ne sont pas toujours optimales en termes de nombre de mouvements, de temps et d'espace. Il est également important de noter que pour parvenir à une solution optimale, le temps de calcul peut s'avérer très long.

FIGURE 1 – Vous trouverez ici les fichiers du projet et du rapport.

8 Listing du programme

```
import heapq
from collections import deque
import time
import os
from random import shuffle

class Taquin:
    def __init__(self, state, parent=None, move=None, g=0):
        self.state = state
        self.parent = parent
```

```
        self.move = move
        self.g = g

    def get_neighbors(self):
            neighbors = []
            moves = [('N', (-1, 0)), ('S', (1, 0)), ('E', (0, 1)), ('W', (0, -1))]
            x, y = None, None

            for i, row in enumerate(self.state):
                for j, cell in enumerate(row):
                    if cell == 0:
                        x, y = i, j
                        break
                if x is not None:
                    break

            for move, (dx, dy) in moves:
                nx, ny = x + dx, y + dy
                if 0 <= nx < len(self.state) and 0 <= ny < len(self.state[0]):
                    new_state = [row.copy() for row in self.state]
                    new_state[x][y], new_state[nx][ny] = new_state[nx][ny], new_state[x][y
]
                    neighbors.append(Taquin(new_state, parent=self, move=move, g=self.g +
1))

            return neighbors

    def get_distance(self, i, j):
        target_x, target_y = divmod(self.state[i][j] - 1, len(self.state[0]))
        return abs(target_x - i) + abs(target_y - j)


    def h(self, heuristic):
        if heuristic == 6:
            return sum(self.get_distance(i, j) for i in range(len(self.state)) for j in
range(len(self.state[0])) if self.state[i][j] != 0)
        else:
            weights = [
                [36, 12, 12,  4,  1,  1,  4,  0],
                [ 8,  7,  6,  5,  4,  3,  2,  1],
                [ 8,  7,  6,  5,  3,  2,  4,  1],
            ]

            if heuristic == 1:
                pi = weights[0]
            elif heuristic in [2, 3]:
                pi = weights[1]
            elif heuristic in [4, 5]:
                pi = weights[2]

            pk = 4 if heuristic in [1, 3, 5] else 1

            return sum(pi[self.state[i][j] - 1] * self.get_distance(i, j) for i in range(
len(self.state)) for j in range(len(self.state[0])) if self.state[i][j] != 0) // pk

    def f(self, heuristic):
        return self.g + self.h(heuristic)

    def __lt__(self, other):
        return self.state < other.state

def solve_taquin(initial_state, final_state, heuristic):
    initial_taquin = Taquin(initial_state)
    frontier = [(initial_taquin.f(heuristic), initial_taquin)]
    explored = set()
    num_explored = 0

    while frontier:
        _, current = heapq.heappop(frontier)

        if current.state == final_state:
            solution = deque()
```

```
            while current.parent is not None:
                solution.appendleft(current.move)
                current = current.parent
            return solution, num_explored

        explored.add(tuple(map(tuple, current.state)))
        num_explored += 1

        for neighbor in current.get_neighbors():
            if tuple(map(tuple, neighbor.state)) not in explored:
                heapq.heappush(frontier, (neighbor.f(heuristic), neighbor))

    return None, num_explored

def generate_random_state(size):
    state = list(range(size * size))
    shuffle(state)
    return [state[i * size:(i + 1) * size] for i in range(size)]

def is_valid_state(state):

    size = len(state)
    flat_state = [cell for row in state for cell in row]
    expected_state = list(range(size * size))
    expected_state[-1] = 0
    inversions = 0

    for i, cell in enumerate(flat_state):
        for j in range(i + 1, size * size):
            if flat_state[j] and flat_state[j] < cell:
                inversions += 1
    if size % 2 == 1:
        return inversions % 2 == 0
    else:
        empty_row = next(i for i, row in enumerate(state) if 0 in row)
        return (inversions + empty_row) % 2 == 1

def generate_states(size):
    initial_state = generate_random_state(size)
    while not is_valid_state(initial_state):
        initial_state = generate_random_state(size)
    final_state = [[(i * size + j + 1) % (size * size) for j in range(size)] for i in
    range(size)]
    return initial_state, final_state

def print_taquin(state):
    for row in state:
        print(' '.join(str(cell) for cell in row))
    print()

if __name__ == "__main__":
    size = int(input("Veuillez entrer la taille du taquin (par exemple, 3 pour un taquin 3
    x3): "))
    initial_state, final_state = generate_states(size)

    print(" tat  initial du taquin :")
    print_taquin(initial_state)

    heuristic = int(input("Veuillez choisir le poids    utiliser (1-6): "))

    start_time = time.time()
    solution, num_explored = solve_taquin(initial_state, final_state, heuristic)
    execution_time = time.time() - start_time

    if solution:
        print(f"Nombre d' tats  explor s: {num_explored}")
        print(f"Nombre de coups jou s: {len(solution)}")
        print(f"Solution pour heuristique h{heuristic}: {solution}")
        print(f"Temps d'ex cution: {execution_time:.2f} secondes")
        print(f"Temps CPU: {os.times()[0]:.2f} secondes")

        current_state = initial_state
```

```
            taquin_instance = Taquin(current_state)
            total_distance = taquin_instance.h(heuristic)
            print(f"Estimation de la distance restante: {total_distance - len(solution)}")
            print(f"Estimation de la fonction heuristique: {total_distance}")

            print("  tat  final du taquin :")
            for move in solution:
                for neighbor in taquin_instance.get_neighbors():
                    if neighbor.move == move:
                        taquin_instance = neighbor
                        break
            print_taquin(taquin_instance.state)
        else:
            print(f"Pas de solution trouv e pour heuristique h{heuristic}.")
```