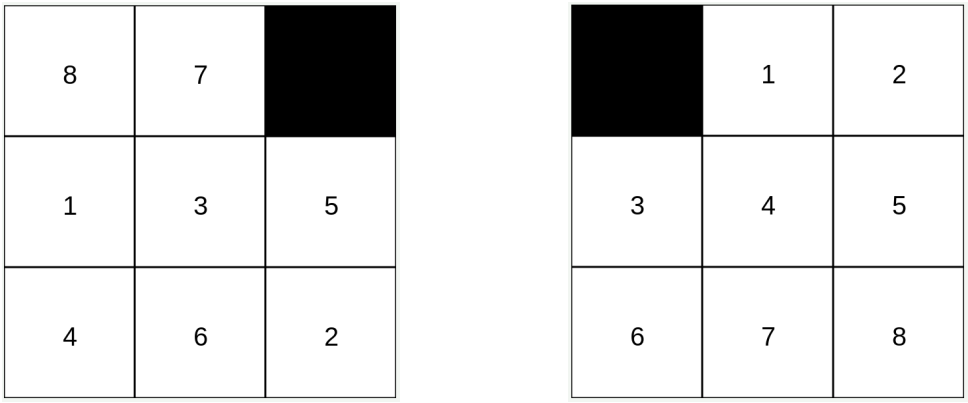
# Résolution de Taquin

Mickaël Péchaud (mickaelpechaud@protonmail.com)

Février 2021




Une position de *taquin* $3 \times 3$, et la position cible à atteindre.

- Documentez vos fonctions.
- Testez au fur et à mesure les fonctions que vous écrivez !

Vous travaillerez dans le fichier *taquin.py* fourni.

## Règles

Le jeu de *taquin* est un jeu à coulissement de pièces, dont le but est de placer les différentes pièces dans une position cible donnée. Un *coup* consiste à déplacer une pièce adjacente à l'espace vide vers cet espace.
Ce TP a pour objet de résoudre des taquins, en un nombre de coups minimal.

## Codage

Dans ce TP, nous allons nous placer dans le cas d'un taquin $3 \times 3$. La position cible est celle de droite sur la figure ci-dessus.
Une position de taquin sera codée par une chaîne de caractères, indiquant les différentes pièces recontrées lorsque l'on parcourt la grille de haut en bas et de gauche à droite.
Ainsi, la position de gauche dans la figure ci-dessus sera représentée par la chaîne `'87 135462'`, et la position cible par `' 12345678'`.
Dans toute la suite, le terme «position» se réfèrera à une telle chaîne de caractères.

## Premières fonctions

1. Compléter la fonction `affiche`, qui permette d'affiche de façon plus agréable une position. Ainsi, `affiche(' 12345678')` devra afficher à l'écran

```
 ---
| 12|
|345|
|678|
 ---
```

2. Compléter la fonction `position_vide`, qui prend en argument une position, et renvoie l'indice où apparaît le caractère *espace* dans la chaîne.
3. Compléter la fonction `intervertit` qui prend en argument une chaîne de caractères et deux indices $i$ et $j$ valides pour cette chaîne, et renvoie une nouvelle chaîne où les lettres d'indices $i$ et $j$ ont été interverties. Vous pourrez utiliser des extractions de tranches.

4. Compléter la fonction `positions_voisines`, qui prend en argument une position, et renvoie la liste des positions accessibles en un coup à partir de cette position.

## Résolution

Nous allons maintenant coder un algorithme de résolution complet.
Pour ce faire, nous allons voir chaque position comme un sommet d'un graphe non orienté. Deux sommets sont reliés par une arête si et seulement si il est possible de passer d'une position à l'autre en un coup.

5. Représenter sous forme de graphe l'ensemble des positions accessibles à partir de la position donnée en exemple dans la figure ci-dessus en deux coups ou moins.
6. Donner une majoration simple du nombre de sommets du graphe entier.

Nous allons parcourir ce graphe des positions jusqu'à ce qu'à atteindre la position cible.
Pour cela, nous allons utiliser deux structures de données.

**L'ensemble des positions actives,** implémenté par une *liste* de sommets `actifs`. Les nouvelles positions seront ajoutées en fin de liste, et la prochaine position à explorer sera la dernière de la liste. La liste est donc utilisée pour implémenter une structure de *pile* des positions actives – ou *LIFO* : **L**ast **I**n **F**irst **O**ut.

**L'ensemble des positions déjà explorées** – i.e. dans `actifs`, ou qui ont été dans `actifs` dans le passé – implémenté par un *dictionnaire* `dejavu`. Les clés seront les positions. Les valeurs seront dans l'immédiat mises à `None` [1].

On rappelle sous forme de pseudocode le principe général d'un algorithme de parcours :

**Pseudocode : Algorithme de parcours**

- Ajouter la position de départ à `actifs` et `dejavu`.
- Tant que `actifs` est non-vide :
  - extraire une position `p` de `actifs`
  - pour chaque position accessible à partir de `p` qui n'est pas dans `dejavu`, rajouter cette position à `actifs` et `dejavu`.

7. À quel type de parcours de graphe correspond l'algorithme décrit ci-dessus ?
8. Compléter la fonction `resoluble`, qui prend en argument une position, parcourt le graphe des positions de la façon décrite ci-dessus et détermine si la position est résoluble – i.e. permet d'attendre la position cible. La fonction renverra un booléen, et affichera de plus le nombre d'itérations effectuées.
   On pourra vérifier que `'123 45678'` est résoluble, mais pas `'213 45678'` [2].
9. Implémenter la fonction `resoluble_recursif`, qui effectue le même travail sans l'affichage du nombre d'itérations en cas d'échec, mais en utilisant une fonction auxiliaire récursive. Tester, et commenter.

On souhaite maintenant calculer un *chemin* résolvant le niveau, c'est-à-dire une liste de positions telle que
- la première position est la position de départ ;
- la dernière est la position cible ;
- on peut passer d'une position de la liste à la suivante en un coup.

Pour ce faire, à chaque position `p` stockée dans le dictionnaire `dejavu`, on associera comme valeur la position «parente» de `p`, c'est-à-dire celle à partir de laquelle `p` a été obtenue.

Une fois la position gagnante atteinte, il suffira donc de remonter successivement les «parentes» jusqu'à la position initiale pour obtenir un chemin.

10. Compléter la fonction `chemin` correspondante. On pourra copier-coller le code de la fonction précédente et l'adapter.

1. Nous pourrions utiliser le type `set` qui permet de coder directement des ensembles. Nous aurons cependant besoin d'avoir un dictionnaire dans les questions suivantes.
2. En travaillant sur le groupe des permutations du taquin, il est possible de trouver une caractérisation assez simple des positions résolubles – testable sans nécessité de parcourir le graphe des positions.

## Plus courts chemins

On souhaite maintenant obtenir un plus court chemin. Nous allons simplement remplacer la structure de *pile* d'`actifs`, où le sommet traité est le dernier entré, par une structure de *file*, où le sommet traité est le premier entré. Pour ce faire, nous allons utiliser le type `collections.deque`, dont les méthodes utiles sont données en annexe. Vous trouverez également un exemple d'utilisation de cette structure dans le fichier.

**11.** Quel est le type de parcours ainsi implémenté ?
**12.** Compléter la fonction `plus_court_chemin` correspondante.

## Statistiques sur l'espace des positions

On souhaite obtenir des informations sur l'espace des positions accessibles à partir de la position cible - par exemple le nombre de coup minimum moyen permettant de résoudre une position.

**13.** Compléter la fonction `distances_a_la_cible`, qui renvoie un dictionnaire
- dont les clés sont les positions accessibles ;
- et dont les valeurs sont les distances de chaque position à la position cible.

*Indication :* on effectuera un parcours en largeur du graphe des positions en partant de la position cible.

**14.** Si l'on note `d` le résultat de la fonction précédente, `d.values()` permet de récupérer la liste des distances à la position cible des positions accessibles.
**(a)** Quel est la distance maximale ? (On pourra utiliser `max` sur une liste.)
**(b)** Quel est la distance moyenne ? (On pourra utiliser `sum` sur une liste.)
**(c)** Tracer l'histogramme des distances. Pour cela, on utilisera les commandes

```
import matplotlib.pyplot as plt
plt.hist(d, max(d))
plt.show()
```

## Exploration guidée

Nous allons maintenant accélérer la vitesse de résolution en utilisant l'algorithme $A^\star$.

Nous allons commencer, pour une position donnée, par obtenir une minoration du nombre de coups nécessaires pour arriver à la position cible. Pour cela, on considère chaque pièce de façon indépendante. Si une pièce est en position $(i, j)$ et doit arriver en position $(i_c, j_c)$ dans la position cible, elle devra être déplacée au minimum $\|(i, j) - (i_c, j_c)\|_1 = |i - i_c| + |j - j_c|$ fois.

Comme une seule pièce est déplacée à chaque coup, la somme de la quantité ci-dessus pour toute les pièces donne une minoration du nombre de coups nécessaires pour atteindre la position cible.

Cette minoration est appelée *heuristique* de la position.

**15.** Vérifier que l'heuristique de la position donnée au début de cet énoncé est 14.
**16.** Compléter la fonction `heuristique`, qui prend en argument une position et renvoie son heuristique.

L'algorithme $A^*$ est une variation sur le principe de l'algorithme de parcours en largeur. Au lieu de parcourir les sommets du graphe des positions de façon concentrique, par distances à la position de départ croissantes, on les parcourt dans l'ordre des $d + h$ croissants, où
- $d$ est la distance du sommet au sommet de départ ;
- $h$ est l'heuristique du sommet.

$d + h$ est appelée *priorité*.

L'heuristique permet de guider l'algorithme dans la direction de la position gagnante, et l'on peut prouver que l'algorithme trouve bien un chemin de longueur minimale.

Voici le pseudocode de l'algorithme $A^\star$.

**Pseudocode : Algorithme $A^\star$**

- Ajouter la position de départ à `actifs` et `dejavu`. Mettre sa distance à 0.
- Tant que `actifs` est non-vide :
  - extraire une position de plus basse priorité `c` de `actifs` ;
  - pour chaque position accessible à partir de `c` qui n'est pas dans `dejavu`, l'ajouter à `actifs` et `dejavu`, et mettre sa distance à celle de $c$ plus un ;
  - pour chaque position accessible à partir de `c` qui est dans `dejavu` et dont la distance est plus grande que celle de $c$ plus un, mettre à jour sa distance.

`actifs` est ici une *file de priorité*, c'est-à-dire une structure sur laquelle on peut effectuer les opérations suivantes :
- tester si elle est **vide** ;
- **ajouter** un élément avec une certaine priorité ;
- **extraire** l'élément de plus basse priorité ;
- **mettre à jour** la priorité d'un élément présent.

Vous trouverez dans le fichier *priorityqueue.py* une implémentation d'une file de priorité [3].

**17.** Compléter la fonction `plus_court_chemin_astar`, qui implémente l'algorithme $A^\star$ pour calculer une solution optimale.
*Indication :* on stockera comme valeurs du dictionnaire des couples (distance, parent).
Tester, et comparer le nombre d'itérations nécessaires pour trouver un plus court chemin avec celle du parcours en largeur. Tester également les temps d'exécution (à l'aide de `time.process_time()`) et commenter.

### Bonus

**18.** Quel est le nombre de positions résolubles ? Reliez le résultat à votre majoration de la question 6. Si vous avez entendu parler de groupe des permutations, généralisez, et prouvez !

**19.** Modifier votre code pour qu'il fonctionne également avec un taquin $4 \times 4$. Cette modification devrait être rapide à effectuer si votre code est bien écrit !

**20.** Pour une heuristique permettant de résoudre plus efficacement des taquin $4 \times 4$, voir http://2.3jachtuches.pagesperso-orange.fr/dossiers/IA/IA.htm. Quelques fonctions permettant de calculer cette heuristique sont disponibles en fin de fichier.

## Annexe

Voici les fonctions et méthodes sur les listes, dictionnaires et deque dont l'usage est autorisé – ainsi que les complexités que vous utiliserez pour les calculs de complexité (en fonction de la longueur de la liste $n$). Tout autre fonction dont vous auriez besoin doit être implémentée !

### Fonctions et méthodes sur les listes

| Opération | Exemple | Complexité |
|---|---|---|
| Création d'une liste vide | `l=[]` | $O(1)$ |
| Accès direct | `l[0]` | $O(1)$ |
| Longueur | `len(l)` | $O(1)$ |
| Concaténation | `l1+l2` | $O(n1+n2)$ |
| Ajout en fin de liste | `l.append(1)` | $O(1)$ |
| Suppression en fin de liste | `l.pop()` | $O(1)$ |
| Extraction de tranche | `l[1 :10]` | $O(n)$, où $n$ est la longueur de la tranche. |
| Répétition | `[0]*k` | $O(n)$, où $n$ est la longueur de la liste créée. |
| Création par compréhension | `[k**2 for k in range(n)]` | $O(n)$ si l'expression est évaluée en temps constant |

3. Le type `queue.priorityQueue` existe en *Python*, mais il n'est pas possible d'y mettre une valeur à jour, ce qui limite son utilisation pour les algorithmes de parcours.

## Fonctions et méthodes sur les dictionnaires

| Opération | Exemple | Complexité |
|---|---|---|
| Création | `d = {cle : valeur}` | $O(1)$ |
| Test d'appartenance d'une clé | `cle in d` | $O(1)$ |
| Ajout d'un couple clé/valeur | `d[cle] = valeur` | $O(1)$ |
| Valeur correspondant à une clé | `d[cle]` | $O(1)$ |

## Fonctions et méthodes sur les deque

| Opération | Exemple | Complexité |
|---|---|---|
| Création | `q = deque()` | $O(1)$ |
| Ajout à la fin | `q.append(e)` | $O(1)$ |
| Suppression au début | `e = q.popLeft()` | $O(1)$ |
| Longueur | `len(q)` | $O(1)$ |