Le Jeu du taquin

PSI 2022/2023

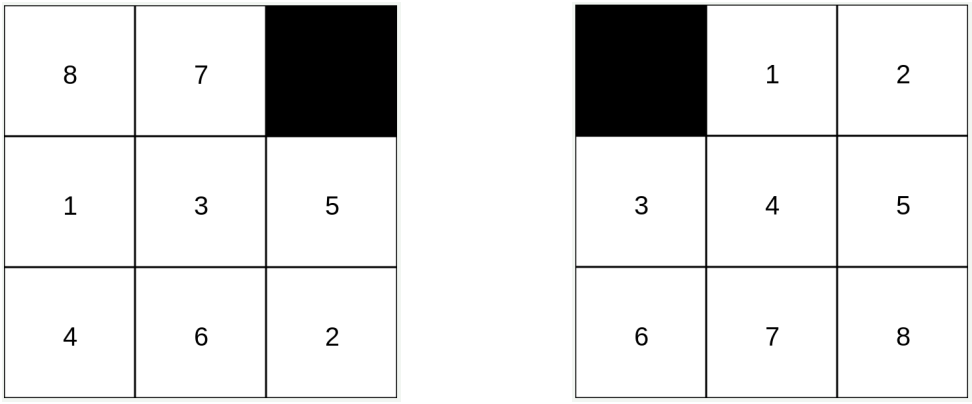

FIGURE 1 – Une position de *taquin* 3×3, et la position cible à atteindre.

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 3×3. 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 permet d'afficher 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 (*slicing*).

❑ **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 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` .

On rappelle ci-dessous le principe général d'un algorithme de parcours :

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`.

❑ **9** — À quel type de parcours de graphe (largeur ou profondeur) correspond l'algorithme décrit ci-dessus ?

❑ **10** — 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'`.

❑ **11** — 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.

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

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.

❑ **13** — Quel est le type de parcours ainsi implémenté ?

❑ **14** — Compléter la fonction `plus_court_chemin` correspondante.