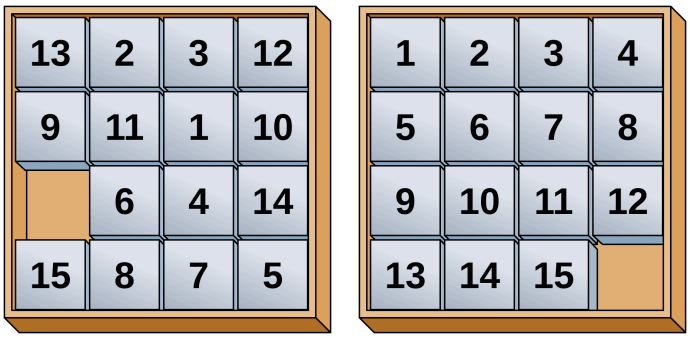
Jeu de taquin

Thèmes : suites récurrentes, matrices, graphes, parcours en largeur

Le *taquin* est un jeu se jouant seul dans lequel le joueur tente de replacer dans l'ordre 15 petites tuiles carrées numérotées de 1 à 15 qui peuvent glisser dans un cadre carré prévu pour 16 tuiles.



Taquin à résoudre Taquin résolu

Fig. 1. – Jeu de taquin (sources : (1) et (2))

Dans ce sujet on s'intéressera à la résolution du **taquin à 8 tuiles** numérotées de 1 à 8 glissant dans un cadre de dimensions 3×3.

Les programmes à écrire et les résultats demandés font intervenir une valeur u0 fixée en début de sujet. Cette valeur peut être différente de celle de vos camarades. Pour vous aider à vérifier vos programmes, les résultats attendus pour la valeur u0 = 42 sont donnés ; sur la fiche réponse vous devrez indiquer les résultats obtenus pour votre valeur de u0.

Attention, pour éviter les incohérences, lorsque vous changez la valeur u0 dans le code il faut réévaluer l'ensemble de votre fichier depuis le départ.

1 Mélange d'un taquin

Un taquin sera codé en `Python` par une liste `t` de 3 sous-listes de longueur 3, représentant chacune une ligne du taquin. Ainsi $t[i][j]$ est la valeur de la tuile à la ligne $i \in \{0, 1, 2\}$ et à la colonne $j \in \{0, 1, 2\}$. L'absence de tuile sera notée par la valeur 0. La tuile de coordonnées $(i, j) = (0, 0)$ sera la tuile en haut à gauche.

Une fonction `print_taquin(taquin: list) -> None` est fournie pour vous permettre d'afficher joliment à l'écran un taquin : `https://cahier-de-prepa.fr/pc-fauriel/download?id=1546` (accès public).

Cette partie se concentre sur la création d'un problème de taquin aléatoire, obtenu en mélangeant aléatoirement un taquin résolu.

Question 1. Écrire une fonction `taquin_résolu()` qui renvoie un nouveau taquin, dans la configuration du taquin résolu, comme représenté à dans la figure de droite ci-dessus, mais avec pour dimensions 3×3. Afficher le résultat à l'écran.

Question 2. Écrire une fonction `place_vide(taquin)` prenant en argument un taquin donné et renvoyant le couple de coordonnées `(i, j)` de la position où il n'y a pas de tuile. Tester cette fonction sur le taquin résolu.

Dans une configuration donnée, il existe au plus 4 déplacements de tuiles possibles ; on les notera `'N'` (nord), `'E'` (est), `'S'` (sud), `'W'` (ouest). Cette direction est le sens de déplacement de la tuile qui se déplace.

Question 3. Écrire une fonction `déplacement(taquin: list, direction: str) -> int` qui
- si le déplacement direction est possible, modifie taquin en conséquence puis renvoie la valeur `1` ;
- si le déplacement n'est pas possible, renvoie la valeur `0` sans changer taquin.

Indication. Commencer par utiliser la fonction `place_vide` pour déterminer la position de la case vide, puis étudier si le déplacement proposé est possible.

On considère la suite $(u_n)_{n\in\mathbb{N}}$ définie par la récurrence suivante :

$$\begin{cases} u_0 &= \texttt{u0} \\ u_{n+1} &= 1022 \times u_n \pmod{2^{30}-3} \end{cases}$$

Question 4. Construire un tableau numpy `u` de longueur 100000 où chaque case `u[i]` contient la valeur de u_i. Donner les valeurs de
a) u_{10},
b) u_{500},
c) u_{10000}.

On note $D(k,\ell)$ une liste de longueur ℓ contenant des déplacements aléatoires. Cette liste est définie ainsi :

$$\forall i \in [\![0, \ell-1]\!], \quad D(k,l)[i] = \begin{cases} \texttt{'N'} \text{ si } u_{i+k} \equiv 0\ [4] \\ \texttt{'E'} \text{ si } u_{i+k} \equiv 1\ [4] \\ \texttt{'S'} \text{ si } u_{i+k} \equiv 2\ [4] \\ \texttt{'W'} \text{ si } u_{i+k} \equiv 3\ [4]. \end{cases}$$

Question 5. Écrire une fonction `D(k: int, l: int) -> list` prenant en arguments les entiers `k` et `l` et renvoyant cette liste de déplacements aléatoires. Donner les valeurs de
a) `D(0, 5)`,
b) `D(50, 5)`,
c) `D(140, 5)`.

Question 6. Écrire une fonction `déplacement_liste(taquin: list, l: list) -> int` prenant en entrée un taquin et une liste de déplacements et effectuant dans l'ordre les déplacements possibles en modifiant `taquin`. Cette fonction renverra le nombre de déplacements réellement effectués.

Donner le nombre de déplacements réellement effectués lorsqu'on part du taquin résolu et qu'on effectue les déplacements suivants :
a) `D(50, 1000)`,
b) `D(3500, 1000)`,
c) `D(7100, 1000)`.

Indication. Utiliser la fonction `déplacement`.

Soit k un entier naturel, on notera $T(k)$ le taquin obtenu en partant du taquin résolu et en appliquant la liste de déplacements $D(k, 1000)$.

Question 7. Écrire une fonction `T(k: int) -> list` renvoyant le taquin aléatoire $T(k)$.
a) Donner $T(700)$.
b) Quelle est la somme des valeurs sur la première ligne de $T(50)$?

c) Quelle est la position de la case vide dans $T(200)$?

2 Graphe du taquin

Les suites de déplacements possibles d'un taquin peuvent être modélisées à l'aide d'un **graphe orienté** dans lequel les sommets sont les configurations possibles du jeu et les arcs les déplacements possibles d'une configuration à une autre par déplacement d'une tuile. Dans une configuration donnée, le but du jeu est alors de trouver un chemin dans le graphe menant de cette configuration à la configuration de taquin résolu. Si possible, on cherchera à obtenir un chemin le plus court possible, c'est-à-dire nécessitant un minimum de déplacements de tuiles.

Afin d'obtenir une numérotation des sommets du graphe, on va associer à chaque taquin un entier unique, appelé ***configuration*** du taquin, et défini de la façon suivante :

$$c(t) = \sum_{i=0}^{2}\sum_{j=0}^{2} t[i][j] \times 10^{3i+j}$$

Question 8. Écrire une fonction `configuration(taquin: list) -> int` renvoyant la configuration d'un taquin donné en entrée. Donner la configuration du :
a) taquin résolu ;
b) taquin $T(50)$;
c) taquin $T(200)$.

Réciproquement, on pourra toujours retrouver un taquin à l'aide de sa configuration en utilisant la fonction réciproque suivante :

```
def décode(c: int) -> list:
    """
    Renvoie l'unique taquin de configuration donnée en entrée
    """
    n = c
    t = [[0, 0, 0], [0, 0, 0], [0, 0, 0]]
    for i in range(3):
        for j in range(3):
            t[i][j] = n % 10
            n = n // 10
    return t
```

Le graphe du jeu de taquin est donc formellement un graphe orienté $G = (S, A)$ où

- l'ensemble des sommets S est l'ensemble des configurations possibles du taquin ;
- l'ensemble des arcs A est l'ensemble des couples (c_1, c_2) où c_1 est une configuration et c_2 une configuration obtenue par un déplacement possible (N, E, S, ou W) depuis c_1.

Un exemple de sommet et ses voisins sont donnés sur la Fig. 2 ci-dessous.

Question 9. Écrire une fonction `voisins(c: int) -> list` prenant en entrée une configuration c et renvoyant la liste des *voisins sortants de* c, c'est-à-dire la liste des configurations accessibles depuis la configuration c en un déplacement de tuile `N`, `E`, `S` ou `W`, en omettant les déplacements impossibles. **On renverra impérativement les voisins dans cet ordre.**

Donner la liste des voisins du sommet
a) correspondant au taquin résolu;
b) correspondant au taquin $T(50)$.

Quel est le degré du sommet de configuration 78120345 ?

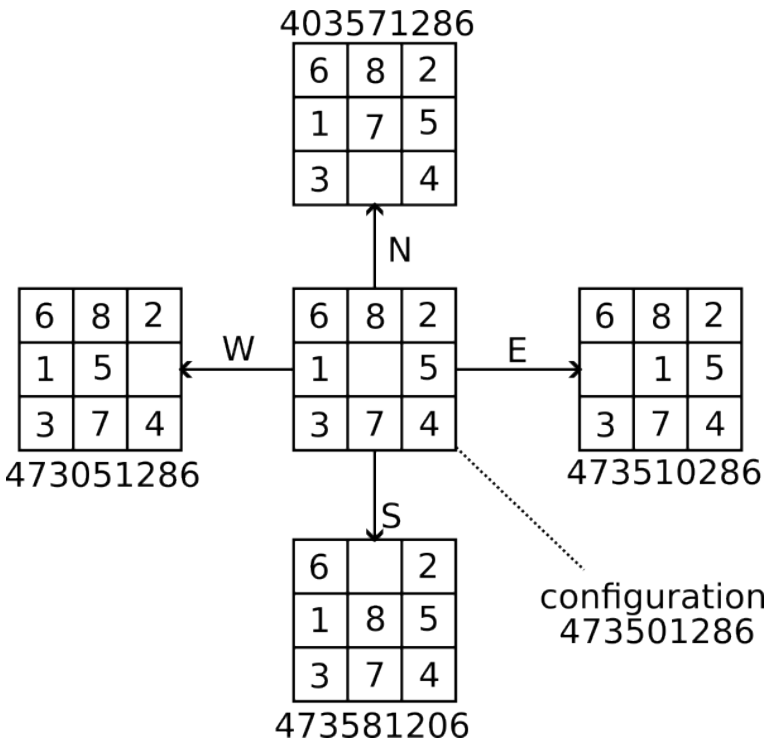


Fig. 2. – une partie du graphe du taquin

Indication. Utiliser la fonction `déplacement` pour tester si les déplacements sont possibles.

On sait que le parcours en largeur permet d'explorer les sommets d'un graphe par ordre de distance en nombre d'arcs depuis le sommet de départ x_0. Autrement dit, on peut se servir du parcours en largeur pour déterminer le chemin le plus rapide pour atteindre la configuration résolue depuis une configuration donnée.

Le parcours en largeur nécessite l'utilisation d'une **file**. On utilisera pour cela l'objet `deque` (*double-ended queue*) de la bibliothèque `collections` de Python :

```
from collections import deque

f = deque() # crée une nouvelle file f
f.append(x) # insère x dans la file f
f.popleft() # extrait un élément de la file f
len(f) # nombre d'éléments dans la file f
```

La file servira à stocker les sommets découverts mais non encore explorés pendant le parcours. Pour se rappeler quels sont les sommets déjà visités, on utilisera un **dictionnaire** dont les **clefs** sont les sommets déjà visités, et la **valeur associée** à chaque clef est le sommet qui a permis sa découverte, c'est-à-dire le sommet qui le précède dans le parcours. On interrompt le parcours lorsqu'on rencontre le sommet correspondant au taquin résolu.

On utilisera donc l'algorithme de parcours en largeur suivant :

PARCOURS EN LARGEUR(c_0 une configuration de départ):

1 initialiser une file vide `f` ;
2 insérer c_0 dans `f` ;
3 initialiser un dictionnaire vide `précédent` ;

```
4  tant que f ≠ ∅ faire
5      extraire le premier élément x de f ;
6      pour chaque y voisin de x faire
7          si y ∉ précédent alors
8              insérer y dans f ;
9              précédent[y] ← x
10         fin si
11     fin pour
12     si x est la configuration résolue alors
13         quitter la boucle tant que
14     fin si
15 fin tant que
16 renvoyer le dictionnaire précédent
```

Question 10. Écrire une fonction `parcours_largeur(c0: int) -> dict` renvoyant le dictionnaire `précédent` qui contient l'ensemble des sommets découverts pendant le parcours, associés à leur sommet précédent dans le parcours.

a) Combien de sommets on été explorés depuis le taquin initial $T(50)$?
b) Combien de sommets on été explorés depuis le taquin initial $T(120)$?
c) Quel sommet précède la configuration résolue dans l'exploration depuis $T(120)$?

Question 11. Écrire une fonction `reconstruction_chemin(c0: int, préc: dict) -> list` qui renvoie un chemin optimal menant de `c_0` à la configuration de taquin résolue, c'est-à-dire qu'elle renvoie la liste des configurations `[c_0, ..., <configuration résolue>]`.

a) Afficher à l'écran les étapes de la résolution de $T(50)$ puis donner les 5 premiers déplacements à effectuer.
b) Afficher à l'écran les étapes de la résolution de $T(120)$ puis donner les 5 premiers déplacements à effectuer.
c) Combien faut-il de déplacements au minimum pour résoudre le taquin $T(100)$?
d) Combien faut-il de déplacements au minimum pour résoudre le taquin $T(200)$?

Indication. Se servir du dictionnaire préc pour reconstruire le chemin en partant de la fin, jusqu'à atteindre `c0`.

3 Sources

a) Par Booyabazooka — `http://en.wikipedia.org/wiki/Image:15-puzzle.svg`, Domaine public, `https://commons.wikimedia.org/w/index.php?curid=1059593`
b) Par Theon — own work, modified from the original file Image:15-puzzle.svg, CC BY-SA 3.0, `https://commons.wikimedia.org/w/index.php?curid=3759824`