Le Jeu du taquin

PSI 2022/2023

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

```
def affiche(position: str):
    '''Affiche une position'''
    s = '|'
    print(' '+3*'-'+' ')
    for i in range(len(position)):
        s += position[i]
        if (i + 1) % 3 == 0:
            print(s+'|')
            s = '|'
    print(' '+3*'-'+' ')
    print()

#test
CIBLE = ' 12345678'
affiche(CIBLE)
```

❑ **2** — Compléter la fonction `position_vide`, qui prend en argument une position, et renvoie l'indice où apparaît le caractère *espace* dans la chaîne.

```
def position_vide(s: str):
    '''Renvoie l'indice de la case vide dans une position'''
    for i in range(len(s)):
        if s[i] == ' ':
            return i
    assert False

# test
position_vide(CIBLE)
position_vide('1234 5678')
```

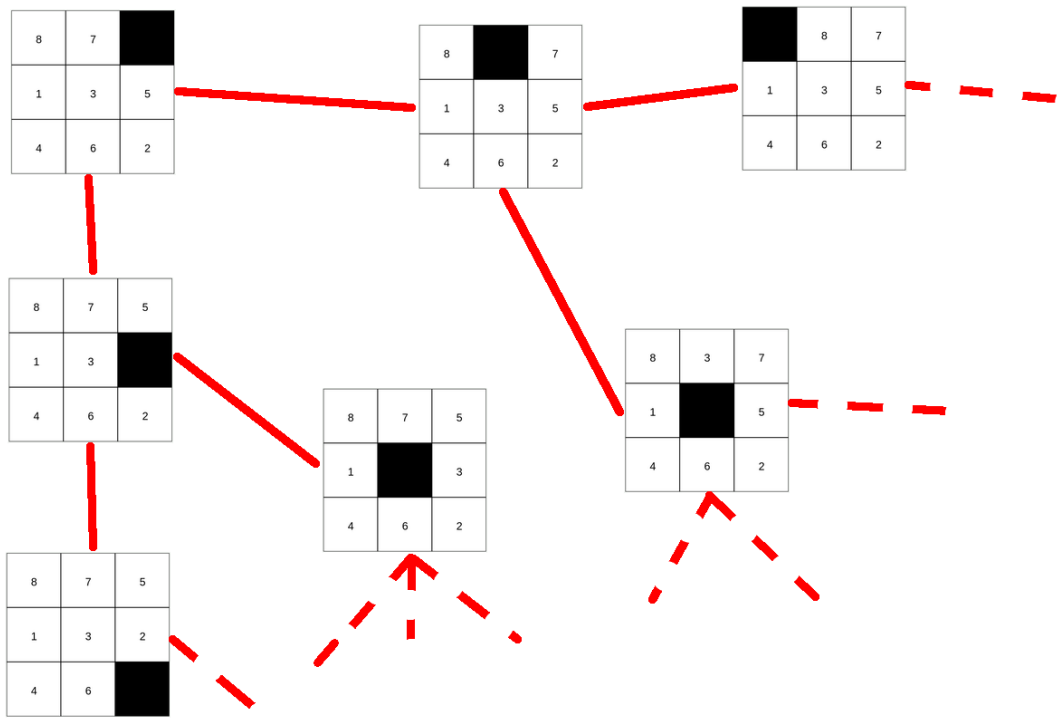
❑ **3** — Compléter la fonction `intervertit` qui prend en argument une chaîne de caractères et deux indices i et j valides pour cette chaîne, et renvoie une nouvelle chaîne où les lettres d'indices i et j ont été interverties. Vous pourrez utiliser des extractions de tranches.

```
def intervertit(s: str, i: int, j: int):
    '''Intervertit les caracteres d'indice i et j dans la chaine de caracteres
   s. Renvoie une nouvelle chaine.'''
    if i < j:
        return s[:i]+s[j]+s[i+1:j]+s[i]+s[j+1:]
    else:
        return s[:j]+s[i]+s[j+1:i]+s[j]+s[i+1:]

# test
s = 'abcdefghij'
print(intervertit(s, 0, 9))
print(intervertit(s, 2, 5))
```

❑ **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.

```
def positions_voisines(position: str):
    '''Renvoie la liste des positions accessibles en un coup a partir de la
   position donnee'''
    n = int(len(position))
    l = []
    i = position_vide(position)
    if i >= 3: #la case vide se deplace vers le haut
        l.append(intervertit(position, i, i-3))
    if i < 6: #la case vide se deplace vers le bas
        l.append(intervertit(position, i, i+3))
    if i % 3 >= 1: #la case vide se deplace vers la gauche
        l.append(intervertit(position, i, i-1))
    if i % 3 <= 1: #la case vide se deplace vers la droite
        l.append(intervertit(position, i, i+1))
    return l

# test
for p in positions_voisines(CIBLE):
    affiche(p)
```

❑ **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.

> Le nombre de sommets est immédiatement majoré par 9!, le nombre de permutations possibles des pièces et de l'espace vide. On peut en fait montrer que son nombre de sommets est $\frac{9!}{2}$.

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

> À un algorithme de parcours en profondeur – les sommets actifs étant stockés dans une structure de pile.

❑ **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'`.

```
def resoluble(position: str) -> bool:
    '''Renvoie vrai si et seulement si la position passee en argument est resoluble'''
    actifs = [position]
    dejavu = {position : None}
    i = 0
    while len(actifs) > 0:
        i += 1
        position_courante = actifs.pop()
        if position_courante == CIBLE:
            print (i, 'iterations')
            return True
        positions = positions_voisines(position_courante)
        for p in positions:
            if not p in dejavu:
                dejavu[p] = None
                actifs.append(p)
    print ('Echec.', i, 'iterations')
    return False

# test
print(resoluble('123 45678'))
print(resoluble('213 45678'))
```

❑ **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.

```
def resoluble_recursif(position: str) -> bool:
    '''Renvoie vrai si et seulement si la position passee en argument est resoluble'''
    dejavu = {position : None}
    def parcours_recursif(position:str) -> bool:
        if position == CIBLE:
            return True
        dejavu[position] = None
        positions = positions_voisines(position)
        for p in positions:
```

```
            if not p in dejavu:
                if parcours_recursif(p):
                    return True
        return False
    return parcours_recursif(position)

# test
print(resoluble_recursif(' 12345678'))
print(resoluble_recursif('213 45678'))
```

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

```
def chemin(position: str):
    '''Renvoie un chemin resolvant la position passee en argument - None si la
   position n'est pas resoluble'''
    actifs = [position]
    dejavu = {position : None}
    i = 0
    while len(actifs) > 0:
        i += 1
        position_courante = actifs.pop()
        if position_courante == CIBLE:
            print (i, 'iterations')
            chemin = []
            while position_courante != None:
                chemin.append(position_courante)
                position_courante = dejavu[position_courante]
            miroir(chemin)
            return chemin
        positions = positions_voisines(position_courante)
        for p in positions:
            if not p in dejavu:
                dejavu[p] = position_courante
                actifs.append(p)
    print ('Echec.', i, 'iterations')
    return None

# test de la fonction chemin
for c in chemin('12 345678'):
    affiche(c)

for c in chemin('123456 78'):
    affiche(c)

```

Plus courts chemins

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

> Il s'agit cette fois-ci d'un parcours en largeur !

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

```python
def plus_court_chemin(position: str) -> list:
    '''Renvoie un plus court chemin resolvant la position passee en argument - None si la position n'est pas resoluble'''
    actifs = deque()
    actifs.append(position)
    dejavu = {position : None}
    i = 0
    while len(actifs) > 0:
        i += 1
        position_courante = actifs.popleft()
        if position_courante == CIBLE:
            print (i, 'iterations')
            chemin = []
            while position_courante != None:
                chemin.append(position_courante)
                position_courante = dejavu[position_courante]
            miroir(chemin)
            return chemin
        positions = positions_voisines(position_courante)
        for p in positions:
            if not p in dejavu:
                dejavu[p] = position_courante
                actifs.append(p)
    print ('Echec.', i, 'iterations')
    return None
#
for c in plus_court_chemin('123456 78'):
    affiche(c)

for c in plus_court_chemin('163425 78'):
    affiche(c)

```