CORRIGÉ DM INFO n°2 : CENTRALE 2017

I. Création d'une exploration et gestion des points d'intérêt

I.A - *Génération d'une exploration d'essai*

I.A.1) Choix de points au hasard

a) Soit `L` la liste des points demandés, initialement vide. On génère au hasard un point de coordonnées entières comprises entre `0` et `cmax`, et s'il n'est pas dans la liste, on l'y ajoute, jusqu'à ce que la liste soit remplie. À la fin, il faut transformer la liste en tableau `numpy`, comme le demande l'énoncé.

```
def generer_Pi(n: int, cmax: int) -> np.ndarray:
    L = []
    while len(L) < n:
        x = random.randint(0, cmax)
        y = random.randint(0, cmax)
        if [x,y] not in L:
            L.append( [x, y] )
     return np.array(L)
```

b) Les arguments de la fonction `generer_Pi` doivent être des entiers naturels ; de plus, le nombre de points distincts possibles est nécessairement inférieur au nombre de points total, c'est-à-dire :

$$\texttt{n} \leqslant (\texttt{cmax}+1)^2$$

(sinon la boucle `while` ne s'arrête pas !).

I.A.2) Choix de points au hasard

Pour simplifier, on peut d'abord écrire une fonction `delta` qui calcule la distance euclidienne entre deux points.

```
def delta(A: np.ndarray, B:np.ndarray) -> float:
    return ((A[0] - B[0])**2 + (A[1]-B[1])**2)**0.5

def calculer_distances(PI: np.ndarray) -> np.ndarray:
    n = len(PI)
    distances = np.zeros( (n+1, n+1), dtype=float )
    pos_actuelle = position_robot()
    for i in range(n):
        for j in range(i):
            distances[i, j] = delta(PI[i], PI[j])
            distances[j, i] = distances[i, j]
            # on peut aussi écrire directement:
            # distances[i, j] = distances[j, i] = delta(PI[i], PI[j])
        distances[i, n] = distances[n, i] = delta(pos_actuelle, PI[i])
    return distances
```

I.B - *Traitement d'image*

I.B.1) Analyse d'une image

La fonction `F1` parcourt l'image et renvoie une liste `h` d'entiers codés sur 64 bits qui compte le nombre de pixels ayant une intensité donnée ; plus précisément, `h[i]` désigne le nombre de pixels de l'image dont l'intensité est égale à `i+n` où `n` est l'intensité minimale.
(Pour ceux qui ont quelques connaissances en photo, il s'agit de l'histogramme...)

I.B.2) Sélection de points d'intérêts

```
def selectionner_PI(photo: np.ndarray, imin:int, imax:int) -> np.ndarray:
    p, q = photo.shape
    L = []
```

```
        for i in range(p):
            for j in range(q):
                if photo[x,y] >= imin and photo[x,y] <= imax:
                # on peut aussi écrire: if imin <= photo[x,y] <= imax:
                    L.append( [i,j] )
        return np.array(L)
```

I.C - *Base de données*

I.C.1)

Un champ de la base de données vaut `NULL` s'il n'est pas rempli. Pour tester si un champ est non rempli, il faut utiliser la condition `IS NULL` (le test « champ = NULL » ne fonctionne pas, car cela revient à comparer deux champs non remplis...)

Dans la requête suivante, pour tester si une exploration est en cours, on teste si elle a une date de début mais pas de date de fin.

```
SELECT EX_NUM FROM EXPLO WHERE EX_DEB IS NOT NULL AND EX_FIN IS NULL
```

I.C.2)

Notons x le numéro de l'exploration concernée.

```
SELECT PI_NUM, PI_X, PI_Y FROM PI WHERE EX_NUM = x
```

I.C.3)

Les coordonnées dans la table `PI` sont en millimètres et le résultat est demandé en m^2...

Dans la solution ci-dessous, on réalise la jointure des deux tables en faisant leur produit cartésien.

```
SELECT PI.EX_NUM, (MAX(PI_X)-MIN(PI_X))*(MAX(PI_Y)-MIN(PI_Y))/1000000 AS Surface
FROM EXPLO, PI
WHERE EXPLO.EXNUM = PI.EXNUM
      AND EX_FIN IS NOT NULL  -- exploration terminee
GROUP BY PI.EX_NUM
```

Si l'on veut faire la jointure des deux tables avec `JOIN`, il faut écrire :

```
SELECT PI.EX_NUM, (MAX(PI_X)-MIN(PI_X))*(MAX(PI_Y)-MIN(PI_Y))/1000000 AS Surface
FROM EXPLO JOIN PI ON  EXPLO.EXNUM = PI.EXNUM
WHERE EX_FIN IS NOT NULL -- exploration terminee
GROUP BY PI.EX_NUM
```

I.C.4)

L'énoncé ne précise pas si les entiers sont codés sur 32 ou 64 bits, signés ou non signés...

Si l'on suppose qu'ils sont codés sur 32 bits et signés, le plus grand entier disponible est alors $2^{31}-1$ et la surface maximale que l'on peut représenter est $(2^{31}-1)^2$ en millimètres carrés, ce qui fait $4.6 \cdot 10^6$ km^2.

I.C.5)

Dans la solution ci-dessous, on réalise la jointure des trois tables en faisant leur produit cartésien.

```
SELECT IN_NUM, COUNT(*), SUM(IT_DUR)
FROM EXPLO AS E, ANALY AS A, INTYP AS I
WHERE E.EX_NUM = A.EX_NUM AND A.TY_NUM = I.TY_NUM
      AND EX_DEB IS NOT NULL AND EX_FIN IS NULL -- exploration en cours
GROUP BY IN_NUM
```

Si l'on veut faire la jointure des trois tables avec `JOIN`, il faut écrire :

```
SELECT IN_NUM, COUNT(*), SUM(IT_DUR)
FROM  INTYP
JOIN ANALY ON EXPLO.EX_NUM = ANALY.EX_NUM
JOIN EXPLO ON EXPLO.EX_NUM = ANALY.EX_NUM
WHERE EX_DEB IS NOT NULL AND EX_FIN IS NULL -- exploration en cours
GROUP BY IN_NUM
```

II. Planification d'une exploration : prmière approche

II.A - *Quelques fonctions utilitaires*

II.A.1) Longueur d'un chemin

```
def longueur_chemin(chemin: list, d:np.ndarray) --> float:
    longueur = 0
    dep = len(d) - 1 # point de départ
    for point in chemin:
        longueur += d[dep, point]
        dep = point
    return longueur
```

II.A.2) Normalisation d'un chemin

```
def normaliser_chemin(chemin:list, n:int) -> list:
    # solution simple, pas optimale car en O(n^2)
    chemin_normalise = []
    # on met chaque première occurence des points du chemin
    for point in chemin:
        if point < n and point not in chemin_normalise:
            chemin_normalise.append(point)
    # on ajoute les points manquants
    for i in range(n):
        if i not in chemin_normalise:
            chemin_normalise.append(i)
    return chemin_normalise

def normaliser_chemin(chemin:list, n:int) -> list:
    # meilleure solution, en O(n)
    # le tableau presents permet de savoir quels sont les points déjà ajoutés
    chemin_normalise = []
    presents = [False] * n
    # on met chaque première occurence des points du chemin
    for point in chemin:
        if point < n and not presents[point]:
            chemin_normalise.append(point)
            presents[point] = True
    # on ajoute les points manquants
    for i in range(n):
        if not presents[i]:
            chemin_normalise.append(i)
    return chemin_normalise
```

II.B - *Force brute*

II.B.1)

Le nombre total de chemin est égal au nombre de permutations de n points, c'est-à-dire $n!$.

II.B.2)

$20! \approx 2 \cdot 10^{18}$ donc la réponse est `non` !

II.C - *Algorithme du plus proche voisin*

II.C.1)

Il est commode d'écrire une fonction auxiliaire qui renvoie le sommet le plus proche d'un sommet donné parmi ceux non encore visités.

```python
def sommet_le_plus_proche(d: np.ndarray, i:int, sommets_visites:list) -> int:
    n = len(d) - 1
    mini = np.inf # plus l'infini
    for j in range(n):
        if j not in sommets_visites and d[i,j] < mini:
            mini = d[i,j]
            imin = j
    return imin

def plus_proche_voisin(d: np.ndarray) -> list:
    n = len(d) - 1
    chemin = [n]
    position = n
    for _ in range(n):
        i = sommet_le_plus_proche(d, position, chemin)
        chemin.append(i)
        position = i
    return chemin
```

II.C.2)

L'algorithme ci-dessus a une complexité en $\mathcal{O}(n^3)$: en effet, la fonction `sommet_le_plus_proche` a une complexité en $\mathcal{O}(n^2)$ à cause du test `if j not in sommets_visites`.
En remplaçant ce test par l'utilisation d'un tableau de booléens comme dans la 2ème solution de **II.A.2**, on peut obtenir une complexité en $\mathcal{O}(n^2)$.

II.C.3)

Il suffit de faire partir le robot du point de coordonnées $(0, 4000)$ ou $(0, 2000)$ pour voir que l'algorithme ne donne pas le chemin le plus court...

III. Deuxième approche : algorithme génétique

III.A - *Initialisation et évaluation*

```python
def creer_population(m:int, d:np.ndarray) -> list:
    population = []
    n = len(d) - 1
    chemin = list(range(n))
    for _ in range(m):
        random.shuffle(chemin)
        longueur = longueur_chemin(chemin, d)
        population.append( [longueur, chemin] )
    return population
```

III.B - *Sélection*

```python
def reduire(p:list) -> None:
    p.sort()
    del p[len(p)//2:]
    # ou bien: p[:] = p[:len(p//2]
```

III.C - *Mutation*

III.C.1)

```python
def muter_chemin(c:list) -> None:
    n = len(c)
    i = random.randint(0, n-1)
    j = random.randint(0, n-1)
    while j == i:
```

```
            j = random.randint(0, n-1)
        # on pouvait aussi faire: i, j = random.sample(list(range(n)), 2)
        c[i], c[j] = c[j], c[i]
```

III.C.2)

```
def muter_population(p:list, proba:float, d:np.ndarray) -> None:
    n = len(p)
    for i in range(n):
        if random.random() < proba:
            chemin = p[i][1]
            muter_chemin(chemin)
            longueur = longueur_chemin(chemin, d)
            p[i] = [longueur, chemin]
```

III.D - *Croisement*

III.D.1)

```
def croiser(c1:list, c2:list) -> list:
    n = len(c1)
    return normaliser_chemin ( c1[:n//2] + c2[n//2:], n )
```

III.D.2)

```
def nouvelle_generation(p:list, d:np.ndarray) -> None:
    n = len(p)
    for i in range(n):
        c1 = p[i][1]
        c2 = p[(i+1)%n][1]
        chemin = croiser(c1, c2)
        longueur = longueur_chemin(chemin, d)
        p.append( [longueur, chemin] )
```

III.E - *Algorithme complet*

III.E.1)

```
def algo_genetique(PI:np.ndarray, m:int, proba:float, g:int) -> float, list:
    # Initialisation
    d = calculer_distances(PI)
    p = creer_population(m, d)

    for _ in range(g):
        # sélection
        reduire(p)
        # croisement
        nouvelle_generation(p, d)
        # mutation
        muter_population(p, proba, d)

    # plus court chemin
    imin = 0
    for i  in range(len(p)):
        if p[i][0] < p[imin][0]:
            imin = i
    return p[imin]
```

III.E.2)

III.E.3)