I Création d'une exploration et gestion de points d'intérêt

I.A – Génération d'une exploration d'essai

1) Choix de points au hasard

a) Première solution, en utilisant la fonction `sample` fournie en annexe (était-elle là pour ça ?) :

```
def generer_PI(n:int, cmax :int) -> np. ndarray :
    zone = []
    for i in range(cmax + 1):
        for j in range(cmax + 1):
            zone.append((i,j))
    return random.sample(zone, n)
```

Solution plus naturelle, mais en utilisant `not in` pour s'assurer que chaque point ajouté est différent des précédents...

```
def generer_PI(n:int, cmax:int) -> np.ndarray:
    L = []
    while len(L) < n:
        x, y = random.randrange(0, cmax + 1), random.randrange(0, cmax + 1)
        if (x, y) not in L:
            L.append((x, y))
    return np.array(L)
```

L'énoncé laisse un voile pudique sur les questions de complexité... Si n est proche du nombre total de points présents dans la zone, on risque de devoir relancer de nombreuses fois le générateur aléatoire pour obtenir les derniers points... Avec pour unité le millimètre, il est probable que n restera très inférieur au nombre total de couples de coordonnées possibles. Sinon, il serait sage de tenir à jour la liste des points non encore sélectionnés et d'effectuer le choix aléatoire parmi ceux-là.

b) Les arguments n et $cmax$ de `generer_PI` doivent être des entiers naturels tels que

$$\boxed{n \leq (1 + cmax)^2.}$$

2) Calcul des distances

Par souci de modularité, j'écris d'abord une fonction qui calcule la distance (euclidienne) entre deux points du plan :

```
def distance(P1:np.ndarray, P2:np.ndarray) -> float:
    x1, y1 = P1
    x2, y2 = P2
    return math.sqrt((x1 - x2)**2 + (y1 - y2)**2)
```

Je remplis le tableau souhaité à l'aide d'une double boucle, en faisant attention à la dernière ligne/colonne puisque c'est la position du robot qui est le point d'indice `n` !

```
def calculer_distances(PI:np.ndarray) -> np.ndarray:
    n = len(PI)
    d = np.zeros((n + 1, n + 1))
    P = position_robot()
    for i in range(n):
        for j in range(i): # on laisse zéro sur la diagonale
            d[i, j] = d[j, i] = distance(PI[i], PI[j])
        d[n, i] = d[i, n] = distance(P, PI[i])
    return d
```

I.B – Traitement d'image

1) Analyse d'une image

Cette fonction détermine le min m et le max b des intensités présentes dans l'image, initialise à zéro un vecteur h dont le nombre de coordonnées est $b - m + 1$, soit le nombre de valeurs possibles pour l'intensité. Ensuite elle parcourt tous les pixels de l'image et incrémente pour chaque pixel d'intensité i le "compteur" $h[i - m]$.

Elle renvoie ainsi un vecteur contenant les nombres de pixels de chaque intensité rencontrée : $h[j]$ contient le nombre de pixels d'intensité $m + j$ où m est l'intensité minimale.

2) Sélection de points d'intérêts

```
def selectionner_PI(photo:np.ndarray, imin:int, imax:int) -> np.ndarray:
    largeur, hauteur = photo.shape
    L = []
    for x in range(largeur):
        for y in range(hauteur):
            if imin <= photo[x, y] <= imax:
                L.append([x, y])
    return np.array(L)
```

I.C – Base de données

1) Il s'agit de sélectionner dans la table `EXPLO` les numéros des explorations pour lesquelles la date de début n'est pas `NULL` tandis que la date de fin est encore `NULL` :

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

2) Pour l'exploration numéro 85 :

```
SELECT PI_NUM, PI_X, PI_Y FROM PI WHERE EX_NUM = 85;
```

3) Il faut ici opérer une jointure des tables `EXPLO` et `PI` selon les numéros des explorations, pour pouvoir d'une part déterminer les explorations terminées (données dans `EXPLO`), d'autre part filtrer les points d'intérêt réellement visités (avec `PI_ARR`) et calculer les aires des rectangles (données dans `PI`).

On divise par 10^6 car les coordonnées sont en mm et l'aire est demandée en m^2.

```
SELECT (MAX(PI_X)-MIN(PI_X))*(MAX(PI_Y)-MIN(PI_Y)) / 1000000 AS surface
FROM PI JOIN EXPLO ON PI.EX_NUM = EXPLO.EX_NUM
WHERE EX_FIN IS NOT NULL AND PI_ARR IS NOT NULL
GROUP BY PI.EX_NUM;
```

Le `GROUP_BY` sert à regrouper les points d'intérêt de chaque exploration de la table `PI`, c'est à chacun de ces groupes que s'appliqueront les fonctions d'agrégation `MIN` et `MAX`.

4) Double jointure ici pour pouvoir restreindre à l'exploration en cours dans la table `EXPLO` (comme au **1)**) les données sur les instruments utilisées, extraites de la table `INTYP` en utilisant transitoirement la table `ANALY` pour récupérer les numéros des types des analyses effectuées durant l'exploration.

```
SELECT IN_NUM, COUNT(*) AS nombre, SUM(IT_DUR) AS durée
FROM INTYP AS i JOIN ANALY AS a ON i.TY_NUM = a.TY_NUM
                JOIN EXPLO AS e ON e.EX_NUM = a.EX_NUM
WHERE EX_DEB IS NOT NULL AND EX_FIN IS NULL
GROUP BY IN_NUM;
```

Ici le regroupement se fait par instrument, pour répondre à la question posée.

II Planification d'une exploration : première approche

II.A – Quelques fonctions utilitaires

1) Longueur d'un chemin

Ne pas oublier de partir de la position courante du robot, dont les distances aux différents points d'intérêt se trouvent en dernière ligne/colonne du tableau `d`.

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

2) Normalisation d'un chemin

Pour éviter de multiplier les recherches dans la liste déjà construite, j'utilise cette fois un "tableau de présence" qui me permet d'éviter les doublons parmi les points d'indice $i \in [\![0, n[\![$. Et l'évaluation paresseuse des booléens me permet d'éliminer les indices $i \geq n$. Après l'épuisement de la liste `chemin`, il n'y a plus qu'à ajouter les indices valides manquants.

```
def normaliser_chemin(chemin:list, n:int) -> list:
    manquant = [True] * n
    chemin_normal = []
    for point in chemin:
        if point < n and manquant[point]:
            chemin_normal.append(point)
            manquant[point] = False
    for point in range(n):
        if manquant[point]:
            chemin_normal.append(point)
    return chemin_normal
```

II.B – Force brute

1) Le nombre de chemins correspond au nombre de permutations des n points d'intérêt, il y en a donc $n!$.

2) Comme $20! \approx 2 \cdot 10^{18}$, en admettant qu'un test de chemin prenne $10^{-9}s$, il faudrait environ 77 ans pour tester tous les chemins. Inutilisable, donc.

II.C – Algorithme du plus proche voisin

1) J'utilise une fonction auxiliaire `indice_proche(p)` qui renvoie un indice correspondant à un point d'intérêt à une distance minimale du point d'indice `p`, non encore visité. Pour cela, je tiens à jour une liste de booléens indiquant les points restant à visiter (afin d'éviter des recherches incessantes dans le début du chemin (cf. la remarque du **I.A–1)b**). Voir le code page suivante.

2) La fonction `indice_proche` a une complexité linéaire en $O(n)$; la boucle de la fonction `plus_proche_voisin` s'exécutant n fois avec un $O(n)$ à chaque tour de boucle et l'initialisation étant en $O(n)$, la complexité de `plus_proche_voisin` est un $O\left(n^2\right)$.

De même `calculer_distances` a pour complexité un $O\left(n^2\right)$.

Finalement, l'algorithme du plus proche voisin a une complexité globale quadratique.

Remarquons que l'utilisation de `in` ou `not in` à la place du tableau de booléens aurait donné une complexité en $O\left(n^3\right)$.

```
def plus_proche_voisin(d:np.ndarray) -> list:
    # fonction auxiliaire
    def indice_proche(p:int) -> int:
        n = len(d) - 1
        m = -1
        for i in range(n):
            if a_visiter[i] and (m < 0 or d[p, i] < d[p, m]):
                m = i
        return m
    # début de plus_proche_voisin
    n = len(d) - 1
    chemin = []
    a_visiter = [True] * n
    a_visiter.append(False)
    position =  n
    while len(chemin) < n:
        position = indice_proche(position)
        chemin.append(position)
        a_visiter[position] = False
    return chemin
```

3) L'algorithme du plus proche voisin est un *algorithme glouton* ne renvoyant pas l'optimum global en général.

Avec les points de coordonnées $A = (0, 0)$, $B = (0, 3\ 000)$ et $C = (0, 7\ 000)$, si le robot se trouve initialement en $P = (0, 2\ 000)$, il va aller en d'abord en B puis en A puis en C et parcourir $1+3+7 = 11$ mètres, alors que le chemin $PABC$ de longueur $2 + 3 + 4 = 9$ mètres est plus court.

III Deuxième approche : algorithme génétique

III.A – Initialisation et évaluation

Il suffit d'utiliser les fonctions déjà écrites...

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

Attention, `shuffle` est du genre "procédure" (renvoie `None`), son appel modifie la liste en place.

III.B – Sélection

J'utilise sans scrupule le tri `p.sort()` donné en annexe. Puisque la première composante des couples est la longueur, l'ordre lexicographique place les chemins les plus courts en tête. Il n'y a plus qu'à supprimer la seconde moitié de la liste avec par exemple la procédure `del` fournie en annexe (on peut aussi répéter `p.pop()` à l'aide d'une boucle...).

```
def reduire(p:list) -> None:
    p.sort()
    del p[len(p) // 2 : ]
```

III.C – Mutation

1) J'utilise à nouveau `random.sample` pour obtenir deux indices distincts.

```
def muter_chemin(c:list) -> None:
    i, j = random.sample(range(len(c)), 2)
    c[i], c[j] = c[j], c[i]
```

2) Nouvel exemple de modularité. Attention à bien utiliser `muter_chemin` comme une procédure.

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

III.D – Croisement

1) J'applique le protocole décrit dans l'énoncé.

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

2) $(i+1)\,\%\,m$ donnera 0 pour $i = m-1\ldots$

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

III.E – Algorithme complet

1) Nouvel exemple frappant de modularité !

```
def algo_genetique(PI:np.ndarray, m:int, proba:float, g:int) -> (float, list):
    d = calculer_distances(PI)
    p = creer_population(m, d)
    for k in range(g):
        reduire(p)
        nouvelle_generation(p, d)
        muter_population(p, proba, d)

    # Recherche du chemin le plus court
    indice_min = 0
    for i in range(1, len(p)):
        if p[i][0] < p[indice_min][0]:
            indice_min = i
    return p[indice_min]
```

2) Le résultat peut se dégrader car on peut faire muter un individu réalisant le minimum à un instant donné. Pour éviter ce problème, on peut décider de ne pas muter l'individu réalisant le minimum d'une génération donnée. Pour cela il suffit, dans la fonction `muter_population`, de remplacer le `range(len(p))` par `range(1, len(p))`. En effet, le début de la liste `p` a été trié lors de l'exécution de `reduire`.

On peut tout aussi facilement conserver les 2 ou 3 meilleurs, mais attention à l'eugénisme : parfois un individu moins performant *a priori* engendre une excellente descendance !

À l'usage, on constate que cet algorithme génétique donne en effet régulièrement une solution meilleure que l'algorithme glouton, mais à condition de laisser passer un nombre de générations plus proche de 100 000 que de 10 000...

3) On peut décider de s'arrêter lorsque :

- on a trouvé la longueur minimale. Avantage : on a résolu le problème. Inconvénient : il fallait connaître la solution, inapplicable donc ! On peut cependant comparer à la valeur renvoyée par `plus_proche_voisin`.
- un certain temps s'est écoulé. Avantage : il suffit d'un chronomètre. Inconvénient : aucune idée sur la précision du résultat.
- le meilleur chemin stagne sur plusieurs générations. Avantage : facile à écrire. Inconvénients : on peut attendre longtemps, possibilité de minimum local. De plus il faut calculer le minimum à chaque étape.
- la population évolue peu. Inconvénients : coûteux à vérifier à chaque étape, aucune garantie d'avoir un minimum voisin de l'optimum.