Centrale - 2017 - Informatique

Rédigé par Jérémy Larochette (Lycée Carnot, Dijon).

On utilise `Python 3`.

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

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

I.A.1) Choix de points au hasard

a)

```python
def generer_PI(n:int, cmax:int) -> np.ndarray:
    """
    - prend en paramètres le nombre de points d'intérêts à générer et la largeur de la
zone d'exploration (supposée carrée)
    - renvoie un objet de type numpy.ndarray contenant les coordonnées de n points
deux à deux distincts choisis au hasard dans la zone d'exploration.
"""
    L = [] # liste des points d'intérêts déjà calculés
    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)
```

ou bien

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

b) Les arguments `n` et `cmax` de `generer_PI` doivent être des entiers naturels tels que $n \leqslant (1+cmax)^2$.

I.A.2) Calcul des distances

```python
def distance(P1:np.ndarray, P2:np.ndarray) -> float:
    "renvoie la distance entre les points de coordonnées P1 et P2"
    x1, y1 = P1
    x2, y2 = P2
    return ((x1 - x2) ** 2 + (y1 - y2) ** 2) ** .5

def calculer_distances(PI:np.ndarray) -> np.ndarray:
    """- prend en paramètre un tableau de n points d'intérêt
       - renvoie un tableau de nombres flottants, de dimension (n + 1) x (n + 1), tel
que l'élément d'indice i,j fournit la distance entre les points d'intérêt i et j,
l'indice n désignant le point de départ du robot."""

    n = len(PI)
    d = np.zeros((n + 1, n + 1)) # tableau des distances
    P = position_robot()

    for i in range(n):
        for j in range(i): # tableau symétrique, nul sur la diagonale
            d[i, j] = d[j, i] = distance(PI[i], PI[j])
        # Cas particulier de la dernière ligne/colonne
        d[n, i] = d[i, n] = distance(P, PI[i])

    return d
```


I.B – Traitement d'image

I.B.1) – Analyse d'une image

Cette fonction parcourt tous les pixels de l'image et incrémente un tableau de compteurs à chaque intensité rencontrée.

Elle renvoie donc un tableau contenant le nombre de pixel de chaque intensité rencontrée (`h[k]` contient le nombre de pixels d'intensité `k + n` où `n` est l'intensité minimale).

I.B.2) – Sélection de points d'intérêts

```python
def selectionner_PI(photo:np.ndarray, imin:int, imax:int) -> np.ndarray:
    """- photo est un tableau représentant une photographie.
    - renvoie un tableau à deux dimensions contenant les coordonnées des points dont
l'intensité sur la photographie est comprise entre imin et imax."""
    longueur, hauteur = photo.shape
    L = []
    for x in range(longueur):
        for y in range(hauteur):
            if imin <= photo[x, y] <= imax:
                L.append((x, y))
    return np.array(L)
```

I.C – Base de données

I.C.1)

```sql
SELECT ex_num FROM explo WHERE ex_deb IS NOT NULL AND ex_fin IS NULL;
```

I.C.2) Pour l'exploration numéro 42 :

```sql
SELECT pi_num, pi_x, pi_y FROM pi WHERE ex_num = 42;
```

I.C.3)

```sql
SELECT (MAX(pi_x) - MIN(pi_x)) * (MAX(pi_y) - MIN(pi_y)) / 1000000
FROM pi
JOIN explo ON pi.ex_num = explo.ex_num
WHERE ex_deb IS NOT NULL AND ex_fin IS NOT NULL; -- exploration terminée
GROUP BY pi.ex_num;
```

I.C.4) On ne peut pas répondre à cette question car on ne sait pas comment sont codés les entiers dans le SGBD.

En faisant l'hypothèse que ces entiers positifs sont stockés en non signé sur 64 bits, il valent au plus $2^{64}-1$ donc la surface maximale sera $\left(2^{64}-1\right)^2 \cdot 10^{-6}$ m^2 soit environ $3{,}4 \cdot 10^{26}$ km^2 : il y a de la marge.

I.C.5)

```sql
SELECT in_num, COUNT(*), SUM(it_dur)
FROM intyp I
JOIN analy A ON I.ty_num = A.ty_num
JOIN explo E ON E.ex_num = A.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 : première approche

II.A – Quelques fonctions utilitaires

II.A.1) Longueur d'un chemin

```
def longueur_chemin(chemin:list, d:np.ndarray) -> float:
    """
    - prend en paramètre un chemin à parcourir et la matrice des distances entre points
d'intérêt
    - renvoie la distance que doit effectuer le robot pour suivre ce chemin en partant
de sa position courante (correspondant à la dernière ligne/colonne du tableau d) et en
visitant tous les points d'intérêt dans l'ordre indiqué.
"""
    longueur = 0
    point_precedent = len(d) - 1
    for point in chemin:
        longueur += d[point_precedent, point]
        point_precedent = point
    return longueur
```

II.A.2) Normalisation d'un chemin

```
def normaliser_chemin(chemin:list, n:int) -> list:
    """
    - prend en paramètre une liste d'entiers
    - renvoie une liste correspondant à un chemin valide, c'est-à-dire contenant une
seule fois tous les entiers entre 0 et n (exclu).
"""
    # tableau de booléen pour tester si un point a déjà été rencontré
    # évite les parcours multiples coûteux de not in
    pas_encore_rencontre = [True] * n
    chemin_normal = []

    # on ajoute les points sans doublon
    for point in chemin:
        # on utilise l'évaluation paresseuse
        if point < n and pas_encore_rencontre[point]:
            chemin_normal.append(point)
            pas_encore_rencontre[point] = False

    # on ajoute les points manquants
    for point in range(n):
        if pas_encore_rencontre[point]:
            chemin_normal.append(point)

    return chemin_normal
```

II.B – Force brute

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

II.B.2) Comme $20! \approx 2 \cdot 10^{18}$, en estimant qu'un test de chemin prenne (minoration très grossière!) 10^{-5} seconde, il faudrait environ 771 500 ans pour trouver le bon chemin. Pas très raisonnable, donc.

II.C – Algorithme du plus proche voisin

II.C.1)

```
def indice_mini(position:int, d:np.ndarray, pas_encore_visite:list) -> int:
    """renvoie un indice de point pas encore visité à distance minimale de
    position (qui est déjà visité), pas_encore_visite est une liste de booléens."""
    n = len(d) - 1
    mini = -1
    for i in range(n):
        if pas_encore_visite[i] and (mini < 0 or d[position, i] < d[position, mini]):
            # le point 'position' étant visité, pas de risque de tomber
            # sur la distance nulle.
            mini = i
    return mini

def plus_proche_voisin(d:np.ndarray) -> list:
    """prend en paramètre le tableau des distances et fournit un chemin d'exploration
en appliquant l'algorithme du plus proche voisin."""

    n = len(d) - 1
    chemin = []
    pas_encore_visite = [True] * n + [False]
    position =  n

    while len(chemin) < n:
        # recherche d'un voisin de position à distance minimale (non nulle)
        position = indice_mini(position, d, pas_encore_visite)
        chemin.append(position)
        pas_encore_visite[position] = False

    return chemin
```

II.C.2) La fonction `indice_min` 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 $O(n) + O(n^2) = O(n^2)$.

Quant à `calculer_distances`, sur le même principe, on obtient comme complexité $O(n) + O(n^2) = O(n^2)$.

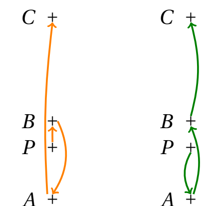
Finalement, l'algorithme du plus proche voisin a une complexité globale quadratique.

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

II.C.3)

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

Avec les points de coordonnées $A = (0,0)$, $B = (0,3000)$ et $C = (0,7000)$, si le robot se trouve initialement en $P = (0,2000)$, 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.



Résultat glouton (à gauche) et résultat optimal (à droite)

III Deuxième approche : algorithme génétique

III.A – Initialisation et évaluation

```python
def creer_population(m:int, d:np.ndarray) -> list:
    """- prend en paramètre le nombre d'individus à engendrer et le tableau des
distances entre points d'intérêt (et la position courante du robot)
    - renvoie une liste d'individus, c'est-à-dire de couples (longueur, chemin)."""
    population = []
    n = len(d) - 1
    points = list(range(n)) # liste de tous les indices de points possibles
    for _ in range(m):
        chemin = random.sample(points, n)
        longueur = longueur_chemin(chemin, d)
        population.append((longueur, chemin))
    return population
```

III.B – Sélection

On trie la liste par ordre lexicographique, donc de longueur d'abord, puis on en supprime la moitié. Ici, il est presque impératif d'utiliser `del` dont l'utilisation est donnée en annexe.

```python
def reduire(p:list) -> None:
    """Modifie p pour n'en garder que la moitié correspondant aux longueurs les plus
petites."""
    m = len(p)
    p.sort()
    del p[m // 2:]
    # ou bien p[:m // 2] = []
    # ou bien utiliser m // 2 fois la méthode p.pop()
```

III.C – Mutation

III.C.1)

```python
def muter_chemin(c:list) -> None:
    """prend en paramètre un chemin et le transforme en inversant aléatoirement deux de
ses éléments."""
    n = len(c)
    i, j = random.sample(range(n), 2)
    c[i], c[j] = c[j], c[i]
```

III.C.2)

```python
def muter_population(p:list, proba:float, d:np.ndarray) -> None:
    """prend en paramètre une population dont elle fait muter un certain nombre
d'individus. Le paramètre proba (compris entre 0 et 1) désigne la probabilité de
mutation d'un individu. Le paramètre d est la matrice des distances entre points
d'intérêt."""
    m = len(p)
    for i in range(m):
        if random.random() <= proba:
            _, chemin = p[i]
            muter_chemin(chemin)
            longueur = longueur_chemin(chemin, d)
            p[i] = (longueur, chemin)
```


III.D – Croisement

III.D.1)

```python
def croiser(c1:list, c2:list) -> list:
    """crée un nouveau chemin à partir de deux chemins passés en paramètre. Ce nouveau
chemin sera produit en prenant la première moitié du premier chemin suivi de la
deuxième moitié du deuxième puis en normalisant le chemin ainsi obtenu."""
    n = len(c1)
    return normaliser_chemin(c1[:n // 2] + c2[n // 2:], n)
```

III.D.2)

```python
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]
        # convient aussi pour i = m - 1
        chemin = croiser(c1, c2)
        longueur = longueur_chemin(chemin, d)
        p.append((longueur, chemin))
```

III.E – Algorithme complet

III.E.1)

```python
def algo_genetique(PI:np.ndarray, m:int, proba:float, g:int) -> (float, list):
    """prend en paramètre :
        - un tableau de points d'intérêts,      - la taille m de la population,
        - la probabilité de mutation proba      - le nombre de générations g.
Cette fonction implante un algorithme génétique et renvoie la longueur du plus court
chemin d'exploration et le chemin lui-même obtenus au bout de g générations."""
    # 1. Initialisation & 2. Evaluation
    d = calculer_distances(PI)
    p = creer_population(m, d)

    for _ in range(g):
        # 3. Sélection
        reduire(p)
        # 4. Croisement
        nouvelle_generation(p, d)
        # 5. Mutation
        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]

    # ou bien, plus simple, (mais moins efficace)
    p.sort()
    return p[0]
```

III.E.2) Le résultat peut se dégrader car on peut muter un individu réalisant le minimum à un instant donné. Pour éviter ce problème, on peut décider de ne pas muter un individu réalisant le minimum (ce qui oblige à le calculer à chaque itération), ou bien de ne muter un individu que si le mutant est meilleur que l'individu lui-même.

III.E.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, pas applicable. On peut cependant comparer à la valeur renvoyer 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. Il faut calculer le minimum à chaque étape.
- La population évolue peu. Inconvénient : coûteux à vérifier à chaque étape. Pertinence du critère ?