CONCOURS BLANC « Centrale - Mines »

Correction

Centrale 2017 - MP-PC-PSI

I Création d'une exploration et gestion des points d'intérêts

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

I.A.1) Choix de points au hasard

a) *Remarque* : cela ne me semble pas une bonne idée de mettre des accents dans les noms de fonctions/variables ...

```
def générer_PI(n:int, cmax:int) -> np.ndarray:
    toutlemonde = [[(i, j) for i in range(cmax + 1)] for j in range(cmax + 1)]
    res=random.sample(toutlemonde, n)
    # Alternative random.shuffle(toutlemonde) puis res=toutlemonde[:n]
    return np.array(res)

# Alternative "à la main"
def générer_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)
```

b) Les deux arguments doivent être des entiers, de plus comme il y a $(cmax+1)^2$ points dans la zone et qu'on veut des points deux à deux distincts on doit avoir $n \leq (cmax+1)^2$.

I.A.2) Calcul des distances

Remarque : créer une fonction distance permettrait sans doute de rendre le code plus clair.

```
def calculer_distances(PI:np.ndarray) -> np.ndarray:
    n = len(PI)
    res = np.zeros((n + 1, n + 1))
    x,y = position_robot()
    for i in range(n):
        x1,y1=PI[i]
        for j in range(i): # car res est symétrique
            x2,y2=PI[j]
            dist= math.sqrt((x1-x2)**2+(y1-y2)**2)
            res[i,j]=dist
            res[j,i]=dist
        # Pour la dernière ligne et colonne
        dist= math.sqrt((x1-x)**2+(y1-y)**2)
        res[i,n]=dist
        res[n,i]=dist
    return res
```

I.B – Traitement d'image

I.B.1) – Analyse d'une image

Cette fonction retourne un tableau `h` où h[k] contient le nombre de fois où l'intensité `n+k` (où `n` est la plus petite intensité de la photo) à été rencontrée.

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

```
def sélectionner_PI(photo:np.ndarray, imin:int, imax:int) -> np.ndarray:
    n, p = photo.shape #n,p=len(photo),len(photo[0])
    L = []
    for x in range(n):
```

```
      for y in range(p):
         if imin <= photo[x, y] and photo[x, y] <= imax:
            L.append([x, y])
   return np.array(L)
```

I.C – Base de données

I.C.1)

Remarque : L'utilisation de `NULL` est hors programme. Je pense que les points ont sans doute été donnés à ceux qui ont mis `=NULL` même si cela ne fonctionne pas : en effet `NULL` correspond à un champ non rempli, tester l'égalité de deux champs non remplis n'a pas grand sens (pour SQL la valeur booléenne de `NULL=NULL` est `unknown`).

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

I.C.2)

Pour l'exploration numéro 42 :

```
SELECT PI_NUM, PI_X, PI_Y FROM PI WHERE EX_NUM = 42;
```

I.C.3)

Noter la conversion en mètres carrés (le AS SURFACE est juste pour donner un nom plus parlant) :

```
SELECT EX_NUM, (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_DEB IS NOT NULL AND EX_FIN IS NOT NULL
GROUP BY PI.EX_NUM;
```

I.C.4)

Soit `max_int` le plus grand entier que l'on peut coder, ainsi la plus grande surface est de $\frac{\texttt{max\_int}^2}{10^6}\,\text{m}^2$.

Si on utilise des entiers non signés sur 64 bits, alors $\texttt{max\_int} = 2^{64}-1$ et la surface est d'alors de $3,4.10^{26}\,\text{km}^2$.

I.C.5)

```
SELECT IN_NUM, COUNT(*) AS NB_UTILISATION, SUM(IT_DUR) AS DUREE_TOT
FROM INTYP
JOIN ANALY ON INTYP.TY_NUM = ANALY.TY_NUM
JOIN EXPLO ON EXPLO.EX_NUM = ANALY.EX_NUM
WHERE EX_DEB IS NOT NULL AND EX_FIN IS NULL
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:
   long = 0
   for k in range(len(chemin)-1):
      long += d[chemin[k], chemin[k+1]]
   return long
```

II.A.2) Normalisation d'un chemin

```
def normaliser_chemin(chemin:list, n:int) -> list:
   res = []
   for point in chemin:
      if point not in res and point<n:
      res.append(point)
   for k in range (n):
      if k not in res:
      res.append(k)
   return res
```

II.B – Force brute

II.B.1)

Il y a $n!$ chemins possible (n choix pour le premier point d'intérêt, $n-1$ pour le second, etc.)

II.B.2)

20! vaut environ $2,4.10^{18}$.

Si on suppose qu'on utilise un ordinateur de 3GHZ, il nous faudrait au minimum un tic d'horloge (en fait beaucoup plus avec nos algorithmes), on aurait besoin de $\frac{20!}{3.10^9}$ seconde soit environ 25.4 années... Ce qui n'est pas très raisonnable.

II.C – Algorithme du plus proche voisin

Remarque : Cet algorithme est ce qu'on appel un « algorithme glouton », il fait à chaque étape le choix d'un optimum local.

II.C.1)

On va écrire une fonction auxiliaire qui détermine le point d'intérêt non visité le plus proche. Pour cette fonction on peut initialiser `dmin` à `np.inf` (qui correspond à $+\infty$) on peut se passer du `or dmin<0` dans la condition.

```
def plus_proche(d:np.ndarray, i:int, visites:list) -> int:
    dmin = -1
    for j in range (len(d)):
        if j not in visites and ( d[i,j]<dmin or dmin <0):
            dmin = d[i,j]
            proche = j
    return proche

def plus_proche_voisin(d:np.ndarray) -> list:
    p = len(d)-1 # la position du robot est dans la dernière colonne
    chemin = []
    while len(chemin)<n:
        p=plus_proche(d,p,chemin)
        chemin.append(p)
    return chemin
```

II.C.2)

La fonction `plus_proche` à une complexité en $\mathcal{O}(n^2)$, en effet on passe n fois dans la boucle et pour un passage dans la boucle on utilise `j not in visites` qui parcours la liste `visites` qui est de longueur au plus n puis on fait des opérations en $\mathcal{O}(1)$.

La fonction `plus_proche_voisin` à une complexité en $\mathcal{O}(n^3)$, en effet on passe n fois dans la boucle et pour chaque passage on utilise la fonction `plus_proche` qui à une complexité de $\mathcal{O}(n^2)$.

La fonction `calculer_distance` à une complexité en $\mathcal{O}(n^2)$, en effet l'appel à `np.zeros` est en $\mathcal{O}(n^2)$, puis on a deux boucles imbriqués où on y fait $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ passages pour y réaliser des opérations en $\mathcal{O}(1)$, ces deux boucles imbriqués ont donc une complexité en $\mathcal{O}(n^2)$.

Il en résulte que l'algorithme a une complexité en $\mathcal{O}(n^3)$.

II.C.3)

Avec les points de coordonnées $A=(0,0)$, $B=(0,3000)$ et $C=(0,7000)$, si le robot se trouve initialement en $P=(0,2000)$ (marche aussi avec $P=(0,4000)$), 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.

Remarque : Sur une copie il est bon de faire un schéma.

III Deuxième approche : algorithme génétique

III.A – Initialisation et évaluation

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

III.B – Sélection

```
def réduire(p:list) -> None:
    m = len(p)
    p.sort() # les chemins sont alors triés selon leur longueur
    del p[m // 2:]
```

III.C – Mutation

III.C.1)

Attention, on doit avoir `i` $\neq$ `j`

```
def muter_chemin(c:list) -> None:
    n = len(c)
    i=random.randrange(0,len(c))
    j=random.randrange(0,len(c))
    while j==i:
        j=random.randrange(0,len(c))
    # Alternative : i, j = random.sample(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:
    for k in range (len (p)):
        if random.random()<proba:
            chemin=p[k][1]
            muter_chemin(chemin)
            p[k]=(longueur_chemin(chemin,d),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)

On peut ne pas séparer le cas où `i=m-1` en mettant dans `c2` : `p[(i+1)%m][1]`.

```
def nouvelle_génération(p:list, d:np.ndarray) -> None:
    m = len(p)
    for i in range(m-1):
        c1, c2 = p[i][1], p[i+1][1]
        chemin = croiser(c1, c2)
        p.append((longueur_chemin(chemin, d), chemin))
    c1, c2 = p[m-1][1], p[0][1]
    chemin = croiser(c1, c2)
    p.append((longueur_chemin(chemin, d), chemin))
```

III.E – Algorithme complet

III.E.1)

Remarque : le `return sorted(p)[0]` est un peu bourrin, on peut programmer cela facilement en $\mathcal{O}(n)$ (alors que `sorted` est en $\mathcal{O}(n\ln(n))$.

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

    for _ in range(g):
        réduire(p)
        nouvelle_génération(p, d)
        muter_population(p, proba, d)
    return sorted(p)[0]
```

III.E.2) C'est tout a fait possible si le meilleur chemin de la liste mute alors la longueur du meilleur chemin peut se dégrader. Pour éviter cela on peut à la nouvelle génération garder le (les) meilleur chemin de la génération précédente.

Une autre méthode serait de lui interdire de muter

III.E.3) On peut décider de s'arrêter lorsque qu'un certain temps s'est écoulé, il a le même avantage (on connait la durée) et le même inconvénient (on ne maitrise pas trop la qualité du résultat) que celui de l'énoncé.

On peut aussi mettre une condition d'arrêt sur la qualité de la famille par exemple sur le fait que le meilleur chemin ne s'améliore pas pendant un certain nombre de générations. L'inconvénient principal est qu'on ne maitrise pas du tout le temps d'exécution. De plus on peut tout a fait avoir un chemin au début pas terrible qui va muter vers un chemin de qualité, mettre une condition d'arrêt portant sur l'homogénéité de la famille peut être intéressant (genre écart entre le meilleur et le pire des chemins).