Centrale Informatique PSI 2017 — Corrigé

Ce corrigé est proposé par Cyril Ravat (professeur en CPGE) ; il a été relu par Julien Dumont (professeur en CPGE) et Jean-Julien Fleck (professeur en CPGE).

Ce sujet d'informatique a pour contexte la mission martienne *Mars Exploration Rovers*. Celle-ci conduit à une étude en trois parties des aspects robotiques de l'exploration, principalement le déplacement du système sur une carte : calculs de distances parcourues et optimisation du chemin reliant tous les points suivant deux approches. De nombreuses questions font intervenir des tableaux `numpy`, donc l'utilisation des fonctions associées, alors que des listes de listes suffiraient la plupart du temps. Un catalogue de ces fonctions est fourni en fin de sujet, avec des explications claires et des exemples.

- La première partie est composée de trois sous-parties indépendantes et de difficultés inégales : la première permet de réaliser deux fonctions élémentaires de génération de carte et de calcul de distance ; la deuxième traite très brièvement de traitement d'image ; la troisième étudie le fonctionnement de la base de données des analyses à effectuer. On regrette la difficulté importante de la première question, qui a dû décourager nombre de candidats, du moins ceux qui ont correctement lu le sujet. Comparativement, les deux questions sur le traitement d'image sont d'une simplicité déconcertante. Enfin, on note que les requêtes SQL demandent l'utilisation d'une syntaxe (`IS NULL`) conceptuellement complexe et certainement peu enseignée.
- La deuxième partie aborde le problème très classique dit « du voyageur de commerce » : il faut optimiser le passage du robot par tous les points d'une liste une seule et unique fois. On met en place ici quelques fonctions élémentaires puis on réalise l'algorithme glouton du plus proche voisin. L'ensemble est équilibré et suffisamment progressif en termes de difficulté.
- La dernière partie continue le travail précédent en proposant pour le même problème l'étude d'un autre type de solution, lui aussi classique : un algorithme génétique. Les questions sont bien détaillées et se suivent parfaitement ; y répondre donne le plaisir de construire petit à petit un algorithme complexe et utilisable sur un cas concret.

Ce sujet laisse au final deux impressions : celle d'un début mal maîtrisé et peu progressif, suivie dans les deux dernières parties d'un sujet classique permettant aux candidats de s'exprimer correctement. Il contient quelques questions de langage SQL mais n'aborde pas la partie ingénierie numérique du programme et ne demande aucune démonstration théorique.

Indications

Partie I

I.A.1.a Le plus simple est de continuer à générer des couples *tant que* le résultat n'en contient pas `n` et d'enregistrer ces couples uniquement s'ils ne sont pas déjà présents dans le résultat, à l'aide de l'opérateur `not in`.

I.A.2 Ce genre de matrices est symétrique, on évite de doubler les calculs. La distance entre deux points se calcule à l'aide du théorème de Pythagore.

I.B Il existe une fonction permettant d'obtenir les dimensions d'un tableau `numpy`.

I.C.1 Pour tester si un champ est égal à `NULL`, ce n'est pas l'opérateur `=` mais l'opérateur `IS` qui est nécessaire : `champ IS NULL` ou `champ IS NOT NULL`.

I.C.3 Un seul résultat par exploration, pour plusieurs explorations, car il s'agit d'une requête d'agrégation. Les champs à calculer et la sélection des explorations terminées se font sur deux tables, une jointure est donc nécessaire.

I.C.4 L'énoncé est incomplet, il ne dit pas comment sont stockés les entiers dans la base. On peut imaginer qu'ils sont positifs et enregistrés sur 64 bits.

I.C.5 Il faut réaliser une jointure. Toutes les tables décrites dans l'énoncé ne sont pas utiles. Les champs `EX_NUM` et `TY_NUM` sont présents et équivalents dans trois tables.

Partie II

II.A.2 Générer une liste de `n` valeurs booléennes permet de savoir rapidement si un point a déjà été visité, sans utiliser la syntaxe `in`.

II.C.1 Il s'agit d'une recherche classique de minimum, mais uniquement sur les points encore disponibles à la $\texttt{i}^{\text{e}}$ itération. Comme en II.A.2, une liste de valeurs booléennes peut aider. Faire attention à l'initialisation du critère minimal recherché.

II.C.3 Les points donnés sont alignés, faire un dessin au brouillon.

Partie III

III.A Pour générer une liste aléatoire d'entiers, le plus simple est d'utiliser la fonction de permutation donnée en fin d'énoncé, sur la liste complète.

III.B Pour trier les chemins, la méthode `sort` fonctionne immédiatement et il n'est pas utile de la coder à la main. Attention, il faut supprimer directement des éléments de la liste `p`, il est interdit d'écrire `p = p[0 :len(p)//2]` par exemple.

III.C.1 La fonction `random.sample` permet d'obtenir plusieurs valeurs distinctes prises aléatoirement au sein d'une liste.

III.C.2 Penser à utiliser les fonctions `muter_chemin` et `longueur_chemin` réalisées aux questions précédentes.

III.D.1 On peut concaténer deux listes avec l'opérateur `+`.

III.E.1 Attention à la syntaxe des fonctions précédentes, notamment de celles qui ne renvoient pas de valeur.

III.E.2 Où est le meilleur chemin dans une génération ? Est-il modifié ?

III.E.3 Classiquement, une série converge si sa valeur n'évolue plus beaucoup.

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

> La syntaxe des définitions de fonctions choisie pour cet énoncé est une possibilité offerte par Python pour rendre le code plus explicite (les « annotations »). Bien que peu d'élèves l'aient déjà vu, son emploi n'a pas dû être un problème.

I.A.1.a Dans la fonction `générer_PI`, on peut choisir de manipuler une liste et d'utiliser la méthode `append` pour ajouter chaque point avant une transformation finale en tableau `numpy`, puisque celle-ci est imposée par l'énoncé :

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

> La difficulté de cette question réside en grande partie dans le besoin de ne garder que des points distincts. La réflexion pour obtenir une liste de n points aléatoire est très courte, mais celle pour éliminer les doublons l'est beaucoup moins, notamment parce qu'un grand nombre de possibilités existent. Si l'on ajoute à cela le retour sous forme de tableau `numpy` exigé, difficulté supplémentaire pour beaucoup de candidats, alors que la liste de listes est suffisante, cette question est plutôt difficile pour un début d'épreuve.
>
> Il est aussi possible d'utiliser un tableau `numpy` depuis le début de l'algorithme, en l'initialisant à un tableau de `n` zéros puis en le remplissant ligne par ligne, en vérifiant que chaque couple tiré au sort n'est pas déjà présent :
>
> ```
> def générer_PI(n:int, cmax:int) -> np.ndarray:
>     PI = np.zeros((n,2),int) # n lignes, 2 colonnes
>     i = 0
>     while i < n:
>         x = random.randrange(0,cmax+1)
>         y = random.randrange(0,cmax+1)
>         if [x,y] not in PI[0:i]:
>             PI[i] = [x,y]; i = i+1
>     return PI
> ```
>
> Malheureusement, malgré ce qu'indique l'annexe de l'énoncé, la construction en `[x,y] in PI[0:i]` ne donne pas le résultat prévu avec un tableau `numpy` :
>
> ```
> >>> PI = np.array([[1,2],[3,4]])
> >>> [1,2] in PI # Normal
> True
> >>> [1,5] in PI # Pas normal !
> True
> ```
>
> ce qui n'est pas vraiment le résultat escompté... À n'en pas douter, la fonction proposée plus haut devrait être acceptée par les correcteurs, mais ce code ne fonctionne pas en pratique. Une des possibilités pour résoudre ce problème, très au-delà de ce que l'on peut attendre d'un candidat, est
>
> ```
> any(np.equal([x,y],PI[0:i]).all(1))
> ```

qui répond correctement sur notre exemple :

```
>>> any(np.equal([1,2],PI).all(1))
True
>>> any(np.equal([1,5],PI).all(1))
False
```

I.A.1.b Les deux nombres doivent **être positifs** et aussi permettre d'obtenir `n` points distincts sur les `cmax+1`, c'est-à-dire

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

Les coordonnées ainsi que `cmax` étant notées en millimètres, il est peu probable que `n` s'approche de cette valeur limite. On peut néanmoins noter dans ce cas que l'algorithme donné à la question précédente devient très inefficace. Il convient alors de le remplacer par un algorithme procédant par élimination d'un nombre complémentaire de points, choisis aléatoirement.

I.A.2 La fonction demandée doit calculer pour chaque couple de points `(i,j)` la distance, à l'aide du théorème de Pythagore. Le résultat est donc symétrique et il faut recopier chaque valeur du triangle inférieur gauche vers le triangle supérieur droit. On n'oublie pas d'ajouter la position actuelle en fin de ligne et de colonne.

```
def calculer_distances(PI:np.ndarray) -> np.ndarray:
    n = len(PI)
    pos = position_robot()
    distances = np.zeros((n+1,n+1))
    for i in range(n):
        for j in range(i):
            distances[i,j] = math.sqrt((PI[i,0]-PI[j,0])**2 \
                                       + (PI[i,1]-PI[j,1])**2)
            distances[j,i] = distances[i,j]
        distances[i,n] = math.sqrt((PI[i,0]-pos[0])**2 \
                                   + (PI[i,1]-pos[1])**2)
        distances[n,i] = distances[i,n]
    return distances
```

On peut éviter une des racines en ajoutant la position actuelle directement à la liste des points d'intérêt avant calcul, mais il faut pour cela être un peu familier de la fonction `np.concatenate`, qui nécessite des listes ou des tableaux `numpy` en argument. Les crochets ici sont essentiels, ce qui n'est pas du tout évident...

```
def calculer_distances(PI:np.ndarray) -> np.ndarray:
    points = np.concatenate((PI,[position_robot()]))
    n = len(points)
    distances = np.zeros((n,n))
    for i in range(n):
        for j in range(i):
            distances[i,j] = math.sqrt(                    \
                              (points[i,0]-points[j,0])**2 \
                             + (points[i,1]-points[j,1])**2)
            distances[j,i] = distances[i,j]
    return distances
```