Concours Centrale filières MP, PC, PSI et TSI : corrigé

Jean-Loup Carré

Informatique commune – 2017

I.A.1.a)
```
def générer_PI(n, cmax):
    R = []
    while len(R) < n:                  # tant qu'on a généré moins de n points
        x = random.randint(0, cmax)    # Générer un point
        y = random.randint(0, cmax)
        if [x,y] not in R:             # Si le point n'a pas déjà été généré
            R.append([x,y])
    return np.array(R)
```

Rapport du jury (filière MP)

« Sélectionner des points deux à deux distincts a perturbé de nombreux candidats. Comme stipulé dans le préambule de l'épreuve, une liste de fonctions utiles était donnée à la fin du sujet. Ceux qui ont su en extraire l'expression « b in a » ont généralement bien surmonté l'obstacle. »

Solution alternative

Il est aussi possible d'adapter l'algorithme S décrit dans *The Art of Computer Programming*, tome 2, de Donald Erwin Knuth.

D. E. Knuth

```
def générer_PI(n, cmax):
    R = np.zeros((n,2))
    N = (cmax+1)**2
    while n > 0:           # Ou alors N > 0
        if random.randint(0, N) <= n:
            R[n-1] = [(N-1)//(cmax+1), (N-1)%(cmax+1)]
            n -= 1
        N -= 1
    return R
```

Dans cette fonction, les couples de nombres sont représentés comme des nombres à deux chiffres en base `cmax+1`.

I.A.1.b) Le nombre de points à générer doit être plus petit que le nombre total de points possibles, c'est-à-dire : $\texttt{n} \leqslant (\texttt{cmax}+1)^2$.
Si cette condition n'est pas vérifiée, la boucle du `while` ne s'arrêtera jamais.

I.A.2) Introduisons une fonction distance qui calcule la distance entre deux points.

```
def distance(A, B):
    return ((A[0]-B[0])**2+(A[1]-B[1])**2)**0.5
```

Pour ne pas avoir à traiter différemment le cas de la position du robot, nous introduisons une fonction `point` qui prend notamment `i` en argument et renvoie le `i`-ème point d'intérêt sauf si `i` vaut `n` : dans ce cas, elle renvoie la position du robot.

```
def point(PI, PR, i, n):
    if i<n :
        return PI[i]
    else :
        return PR
```

On peut alors écrire la fonction finale.

```
def calculer_distances(PI):
    PR = np.array(position_robot())
    n = len(PI)
    Dist = np.zeros([n+1, n+1])
    for i in range(n+1):
        for j in range(n+1):
            Dist[i, j] = distance(point(PI, PR, i, n), point(PI, PR, j, n))
    return Dist
```

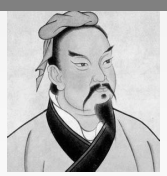
I.B.1) Cette fonction compte le nombre d'occurrences de chaque valeur présente dans l'image. Elle renvoie une liste h telle que pour toute valeur p présente sur la photo, $h[p-n]$ (où n est la valeur minimale présente sur la photo) est le nombre d'occurrences de p.

I.B.2)
```
def sélectionner_PI(photo, imin, imax):
    PI = []
    n, p = photo.shape
    for i in range(n):
        for j in range(p):
            if imin <= photo[i, j] <= imax:
                PI.append([i, j])
    return np.array(PI)
```

Remarque

Dans les questions suivantes, le sujet manipule des valeurs `NULL`, ce qui est largement **hors-programme** en CPGE.
Une valeur `NULL` représente un champ vide, non-rempli. La condition `NULL = NULL` teste si un champ non rempli vaut un autre champ non rempli. Il est difficile d'attribuer une valeur *true* ou *false* à cette condition, c'est pourquoi SQL a trois valeurs de vérité, *true*, *false* et *unknown* (Cf la norme SQL, partie II, section *Boolean types*). La condition `NULL = NULL` renvoie *unknown*.
C'est pourquoi sont mis à disposition `IS NULL` et `IS NOT NULL` qui permettent de vérifier si une valeur vaut `NULL`.

Conseil stratégique


Sun Tsu

Lorsqu'une question hors-programme est posée, répondez-y au mieux avec ce que vous avez appris. Ainsi, il est probable que le jury ait compté juste les réponses utilisant des conditions de la forme `X = NULL` au lieu de `X IS NULL`.

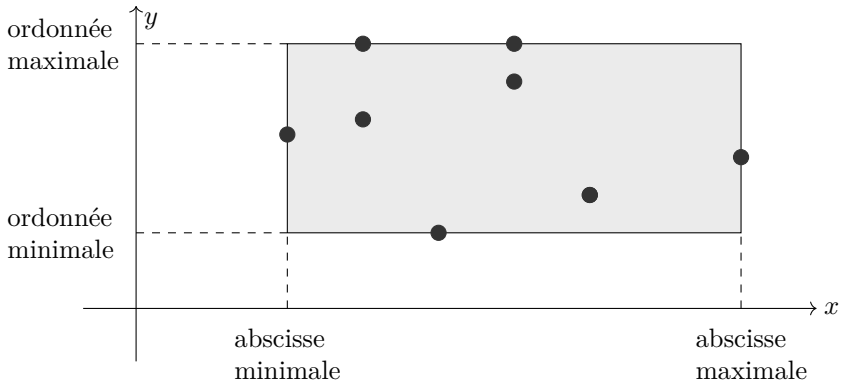
I.C.1) L'exploration en cours a une date de début, mais pas de date de fin. D'où la requête suivante.

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

I.C.2) Si on connait le numéro de l'exploration voulue, il suffit de remplacer dans la requête suivante `XXX` par cette valeur pour avoir la liste des points d'intérêts recherchés.

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

I.C.3) Représentons un exemple sur un dessin. Les disques représentent les points d'intérêts.



La formule de l'aire du rectangle est `(MAX(PI_X)-MIN(PI_X)) * (MAX(PI_Y)-MIN(PI_Y))`.
On écrit la requête voulue :

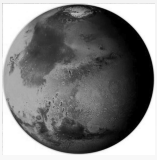
```
SELECT EX_NUM, (MAX(PI_X)-MIN(PI_X)) * (MAX(PI_Y)-MIN(PI_Y)) AS SURFACE
  FROM EXPLO JOIN PI ON EXPLO.EX_NUM=PI.EX_NUM
  WHERE EX_FIN IS NOT NULL      -- La zone est déjà explorée
  GROUP BY EX_NUM
```

I.C.4) Comme les coordonnées sont positives (Cf page 2 du sujet), une exploration a tous ses points dans un carré de sommets $(0,0)$, $(0,\texttt{max\_int})$, $(\texttt{max\_int},\texttt{max\_int})$ où `max_int` est l'entier maximal représentable, soit une superficie totale de $\frac{\texttt{max\_int}^2}{10^6}\text{m}^2$.
La valeur exacte de `max_int` dépend de la représentation choisie pour les entiers. Si les entiers sont représentés par des entiers signés de 32 bits (une représentation courante), alors `max_int` vaut $2^{31}-1 = 2\,147\,483\,647$ soit une superficie maximale d'approximativement $4.6\times10^{12}\text{m}^2$ soit 4.6 millions de km^2.
Avec des entiers non-signés sur 64 bits, alors $\texttt{max\_int} = 2^{64}-1 \approx 1.8\times10^{19}$ ce qui fait une superficie totale d'environs $3.4\times10^{32}\text{m}^2$.

Rapport du jury (filière PC)

« La question 4 nécessitait de prendre des initiatives en définissant le nombre de bits de codage des entiers pour en déduire la surface maximale d'exploration enregistrable. »

Culture générale martienne


Mars

La superficie de Mars est de 114.8 millions de km^2. Ainsi, avec des entiers signés de 32 bits, on peut représenter 4% de la surface martienne.

En outre, avec des entiers non-signés de 64 bits, la surface d'une zone d'exploration peut couvrir toute la surface de Mars. Néanmoins, si on utilise une seule zone pour tout Mars, on ne peut plus négliger la courbure de la planète.

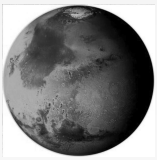
I.C.5) On fait un groupe par instrument. On identifie les instrument par leur numéro (vu que c'est la clef primaire de la table des instruments).

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

La table `EXPLO` permet de se limiter à l'exploration en cours.

Culture générale martienne

Utiliser des unités exotiques (des unités hors du sytème international d'unités) comme ici le jour martien ou le millimètre est source d'erreurs. Par exemple, la sonde Mars Climate Orbiter s'est crashée sur Mars à cause d'un problème de conversion d'unités (des livres [a] secondes en newtons secondes). Voici un extrait du rapport (1999) sur le crash.


Mars

« *The MCO MIB has determined that the root cause for the loss of the MCO spacecraft was the failure to use metric units in the coding of a ground software file, "Small Forces," used in trajectory models. Specifically, thruster performance data in English units instead of metric units was used in the software application code titled SM_FORCES (small forces). The output from the SM_FORCES application code as required by a MSOP Project Software Interface Specification (SIS) was to be in metric units of Newton-seconds (N-s). Instead, the data was reported in English units of pound-seconds (lbf-s).* »

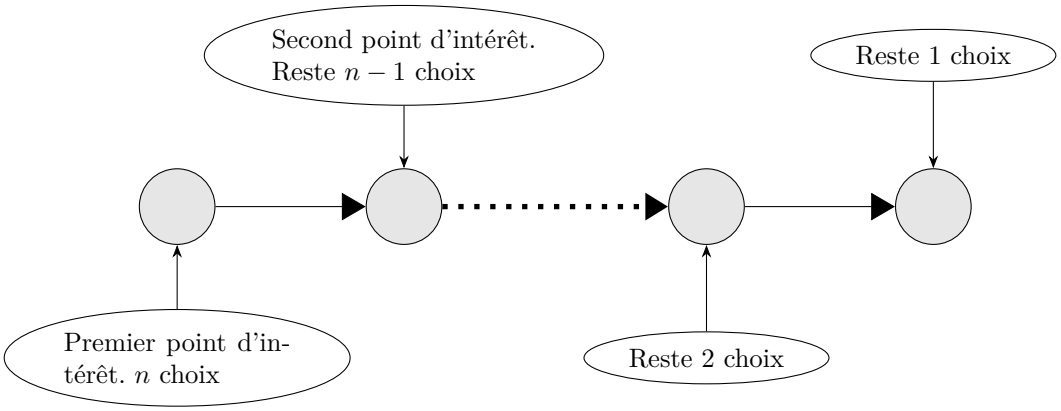
Certains langages, par exemple F# (une variante du langage OCaml enseigné en option informatique en MPSI/MP), permettent de vérifier à la compilation s'il n'y a pas d'erreur d'unités.

a. Il s'agit de la livre-force, qui mesure une force, à ne pas confondre avec la livre (unité de masse).

II.A.1)
```
def longueur_chemin(chemin, d):
    long = 0
    for k in range(len(chemin) - 1):
        long += d[chemin[k], chemin[k+1]]
    return long
```

II.A.2)
```
def normaliser(chemin, n):
    valide = []
    for point in chemin:
        if point not in valide:
            valide.append(point)
    for k in range(n):
        if k not in valide:
            valide.append(k)
    return valide
```

II.B.1) Pour le premier point d'intérêt, nous avons n choix, pour le second il reste $n-1$ choix (n'importe quelle valeur entre 0 et n sauf la valeur déjà choisie), etc.



In fine, nous avons, pour un chemin, $n!$ choix. Il y a $n!$ chemins différents passant exactement une fois par n points.

II.B.2) $20! \approx 2.4 \times 10^{18}$ ce qui est beaucoup.

Si on effectuait le calcul sur un processeur tournant à 2GHz, il nous faudrait a minima un tic d'horloge (très large sous-approximation, en pratique il faut beaucoup plus) pour traiter un chemin, il faudrait donc bien plus de $\frac{20!}{2\times10^9} \times \frac{1}{60} \times \frac{1}{60} \times \frac{1}{24} \times \frac{1}{365.25} \approx 38$ ans pour faire le calcul.

Cette méthode n'est pas utilisable pour une zone d'exploration contenant 20 points d'intérêts.

Rapport du jury (filière PSI)

« ***II.B.2)*** *Bien peu de candidats justifient convenablement (référence au nombre d'opérations par seconde pour un processeur) le fait que l'algorithme n'est pas utilisable dans ce cas.* »

Rapport du jury (filière PC)

« *La plupart des candidats trouvent le nombre de chemins possibles mais un minimum de justification était attendu. Pour conclure, il fallait non seulement réaliser une application numérique pour déterminer l'ordre de grandeur du nombre d'itérations à prévoir, mais aussi pour comparer ce nombre aux puissances de calcul des processeurs actuels.* ».

II.C.1) Commençons par introduire une fonction auxiliaire `plus_proche` qui prend en argument la matrice `d` des distances, un sommet `i` et la liste des sommets déjà visités et qui renvoie le sommet le plus proche de `i` qui n'a pas encore été visité.

```
def plus_proche(d, i, visités):
    dmin = np.inf
    for j in range(len(d)):
        if j not in visités and d[i, j] < dmin:
            dmin = d[i, j]
            proche = j
    return proche
```

On peut alors écrire la fonction demandée.

```
def plus_proche_voisin(d):
    p = len(d) - 1  # position initiale du robot
    chemin = [p]
    for k in range (len(d)-1):
        p = plus_proche(d, p, chemin)
        chemin.append(p)
    return chemin
```

II.C.2) Concernant la fonction `calculer_distance` :

- l'appel à la fonction `zeros` est en $\mathcal{O}(n^2)$ vu qu'on écrit $(n+1)^2$ zéros
- on passe dans chaque `for` $n+1$ fois, le corps du `for` intérieur est en $\mathcal{O}(1)$.

Sa complexité est donc en $\mathcal{O}(n^2) + (n+1) \times (n+1) \times \mathcal{O}(1) = \mathcal{O}(n^2)$.

Concernant la fonction `plus_proche` :

- `visité` est de longueur au plus n, donc `j not in visités` est en $\mathcal{O}(n)$.
- Le reste du corps de la boucle `for` est en $\mathcal{O}(1)$.
- On passe n fois dans la boucle `for`.

Sa complexité est donc en $\mathcal{O}(1) + n \times (\mathcal{O}(n) + \mathcal{O}(1)) = \mathcal{O}(n^2)$.

Concernant la fonction `plus_proche_voisin` :

- Le corps du `for` est en $\mathcal{O}(n^2)$ à cause de l'appel à `plus_proche`.
- On passe n fois dans le `for`.

Sa complexité est en $\mathcal{O}(n^3)$

En conclusion, la complexité de l'algorithme est en $\mathcal{O}(n^2) + \mathcal{O}(n^3) = \mathcal{O}(n^3)$.

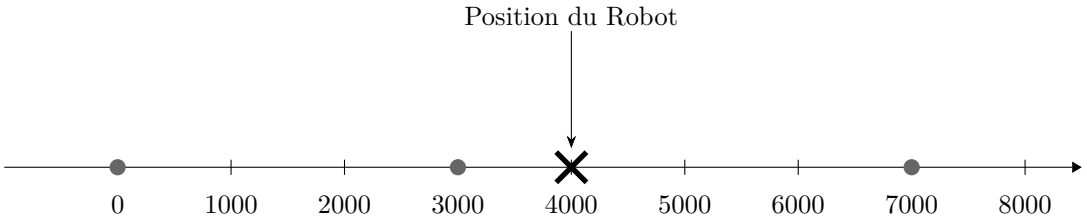
Attention ! Piège !

Ne pas oublier la complexité du `j not in visités`.

Rapport du jury (filière MP)

« Les complexités sont trop souvent données sans justification ; La complexité linéaire cachée dans l'expression « x in list » a échappé à beaucoup. ».

II.C.2) On représente sur le shéma suivant les points d'intérêts (l'axe des ordonnées a été représenté horizontalement) et on choisit comme position initiale du robot le point de coordonnées $(0, 4000)$.



Ainsi, le chemin donné par l'algorithme sera $(0, 3000), (0, 0), (0, 7000)$, ce qui fait une distance totale de 11 000 mm tandis que le chemin le plus court est $(0, 7000), (0, 3000), (0, 0)$ pour une distance totale de 10 000 mm.

Rapport du jury (filière PC)

« Beaucoup de candidats ont trouvé un exemple où l'algorithme ne fournit pas le plus court chemin. Un schéma était souvent plus clair qu'un long discours à cette question. »

III.A) On définit une fonction auxiliaire qui crée un individu.

```
def créer_individu(d):
    chemin = list(range(len(d)-1)) # -1 à cause de la position du robot
    random.shuffle(chemin)
    long = longueur_chemin(chemin, d)
    return [long, chemin]
```

Puis on écrit la fonction permettant de créer une population.

```
def créer_population(m, d):
    L = []
    for k in range(m):
        L.append(créer_individu(d))
    return L
```

III.B)
```
def réduire(p):
    p2 = sorted(p)
    p[:]=p2[:len(p)//2]
```

Remarque

Le sujet ne précisant pas s'il est important de garder l'ordre relatif des éléments, nous ne le conserverons pas. Ne soyons pas plus royaliste que le roi.

III.C.1)
```
def muter_chemin(c):
    i = random.randint(0, len(c)-1)
    j = random.randint(0, len(c)-2)
    if j>=i:
        j +=1
    c[i], c[j] = c[j], c[i]
```

Pour que j soit différent de i, on tire j dans un ensemble ayant une valeur de moins et on décale toutes les valeurs supérieures à i de 1.

Rapport du jury (filière PSI)

« ***III.C.1)*** *On a souvent vu un oubli d'avoir 2 points distincts.* »

III.C.2) On introduit d'abord une fonction permettant de faire muter un individu.

```
def muter_individu(I, d):
    muter_chemin(I[1])
    I[0] = longueur_chemin(I[1], d)
```

Puis on itère la fonction précédente pour faire muter une population.

```
def muter_population(p, proba, d):
    for k in range(len(p)):
        if random.random()<proba:
            muter_individu(p[k], d)
```

III.D.1)
```
def croiser(c1, c2):
    n = len(c1)
    return normaliser(c1[:n//2] + c2[n//2:],n)
```

III.D.2) On introduit d'abord une fonction permettant de croiser deux individus.

```
def croiser_individus(i1, i2, d):
    c = croiser(i1[1], i2[1])
    return [longueur_chemin(c, d), c]
```

Puis on itère la fonction précédente pour croiser une population.

```
def nouvelle_génération(p, d):
    n = len(p)
    for k in range(n):
        p.append(croiser_individus(p[k], p[(k+1)%n], d))
```

III.E.1) L'algorithme final.

```
def algo_génétique(PI, m, proba, g):
    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]
```

Remarque

Le `sorted` final coûte du $\mathcal{O}(n\ln(n))$ alors qu'on peut faire la même chose (extraire le minimum) en $\mathcal{O}(n)$. Cependant, optimiser cette partie du programme n'aurait que peu d'intérêt car elle ne représente qu'une faible partie du temps de calcul.

III.E.2) Oui, c'est possible : si le meilleur chemin mute, alors la longueur du meilleur chemin peut baisser.
Pour que cela ne puisse plus arriver, on peut réduire à zéro la probabilité que le meilleur chemin de l'ancienne génération mute (et laisser la même probabilité pour les autres chemins). Il suffit, dans notre code, de changer le `range(len(p))` de la fonction `muter_population` en `range(1, len(p))` (car l'ancienne génération est triée par notre fonction `réduire`).

III.E.3) On peut alors envisager comme conditions d'arrêts :

a) Le meilleur chemin ne s'améliore pas pendant X générations.
b) Le pire chemin sélectionné ne s'améliore pas pendant X générations.
c) L'écart entre le meilleur et le pire chemin sélectionné ne dépasse pas x%.

Ces trois solutions ont comme défaut que le temps de calcul n'est pas à priori borné (au contraire de la solution retenue dans le sujet). Par contre elles s'arrêtent dès qu'on arrive sur une solution "difficilement améliorable"

La solution a) nous dit de nous arrêter quand la meilleure solution de progresse plus, mais peut-être que dans le reste de la population est en train d'émerger une autre solution plus performante. La b) nous fait nous arrêter quand les solutions secondaires ne progressent plus, mais peut-être que la meilleure solution s'améliore encore. On peut, en ralentissant le temps de convergence, ne s'arrêter que si on a a) et aussi b).

Le c) assure qu'on est arrivé à une situation homogène, où toute la population propose des solutions comparables.