Corrige

1 Machines à café connectées

1.1 Requêtes sur la base de données

Q. 1 `SELECT COUNT(*) FROM clients;`

Q. 2 `SELECT id_client, AVG(note) FROM retours GROUP BY id_client;`

Q. 3 `SELECT id_client, AVG(note) FROM retours GROUP BY id_client HAVING AVG(note) <= 2;`

Q. 4 `SELECT nom, email, AVG(note) FROM clients JOIN retours ON id=id_client GROUP BY id_client HAVING AVG(note) <= 2;`

1.2 Interpolation des données par la méthode des k plus proches voisins

Q. 5

```
def dist(p, q):
    return ((p[0] - q[0]) ** 2 + (p[1] - q[1]) ** 2) ** 0.5
```

Q. 6

```
def plus_proche(p, l):
    dmin = +float('inf')
    pp = None
    for x in l:
        d = dist(x, p)
        if d < dmin:
            dmin = d
            pp = x
    return pp
```

Q. 7 on écrit `list(note.keys())` ou

```
l = []
for k in note:
    l.append(note[k])
```

Q. 8 Écrire une fonction `note_prédite_1NN(g: float, r: float, note: dict) -> int` faisant intervenir les fonctions précédentes et mettant en œuvre la méthode des k plus proches voisins avec $k = 1$.

```
def note_prédite_1NN(g, r, note):
    l = list(note.keys())
    pp = plus_proche((g, r), l)
    return note[pp]
```

Q. 9 Pour cette question le plus facile est d'extraire k fois le plus proche.

```
def plus_proches(p, l, k):
    assert(len(l) >= k)

    res = []
    for i in range(k):
        pp = plus_proche(p, l)
        l.remove(pp)
        res.append(pp)
    return res
```

La complexité est alors en $O(nk)$. Une autre solution était de trier la liste de points par distance croissante à `p`. La complexité était alors en $O(n \log n + k)$.

Q. 10

```
def note_prédite_kNN(g, r, note, k):
    l = list(note.keys())
    pps = plus_proches((g,r), l, k)
    # On va maintenant compter les valeurs pour savoir la plus frequente
    compte = [0 for i in range(21)]
    for x in pps:
        compte[note[x]] += 1
    return argmin(compte)
```

où `argmin` est une fonction facile à écrire qui retourne l'indice de la valeur maximale d'une liste.

11. On peut commencer par s'occuper de l'identifiant du joueur autre que celui qui a le trait, puis construire position par position la liste des positions accessibles en un (demi-)coup pour le joueur ayant le trait.

```
def successeurs(i, j, u):
    if u == 'A':
        v = 'B'
    else:
        v = 'A'

    liste = []
    for k in range(1, i):
        liste.append((i-k, j, v))
    for k in range(1, j):
        liste.append((i, j-k, v))
    for k in range(1, min(i,j)):
        liste.append((i-k, j-k, v))

    return liste
```

12. $\Omega_A = \{(0,0,A)\}$ et $\Omega_B = \{(0,0,B)\}$.

13. L'ensemble Attr_k est l'ensemble des positions où le joueur A peut gagner en au plus k (demi-)coups.

14. On obtient

$$\begin{aligned} \mathrm{Attr}_0 &= \{(0,0,A)\}, \\ \mathrm{Attr}_1 &= \mathrm{Attr}_0 \cup \{(0,1,B),\ (1,0,B)\}, \\ \mathrm{Attr}_2 &= \mathrm{Attr}_1 \cup \{(1,1,A)\} \text{ et} \\ \forall k \geqslant 3,\ \mathrm{Attr}_k &= \mathrm{Attr}_2. \end{aligned}$$

    Le joueur A a une stratégie gagnante : elle consiste, au début de la partie, à retirer la bille ou la pièce de son sac, mais pas les deux. Cela amène B dans une position où il n'a pas d'autre choix que de retirer l'unique autre élément, et donc aboutit à une position finale gagnée pour A.

15. TODO

16. On obtient

$$F(2,3) = 7,$$
$$F(3,2) = 7,$$
$$F(3,3) = 9.$$

17. On additionne les poids individuels des arêtes:

```
def poids_chemin(c):
    p = len(c) - 1
    s = 0
    for k in range(p):
        (i, j) = c[k]
        (a, b) = c[k+1]
        s += P[i, j, a, b]
    return s
```

Variante avec une écriture en compréhension:

```
def poids_chemin(c):
    p = len(c) - 1
    return sum(P[c[k][0], c[k][1],
                 c[k+1][0], c[k+1][1]]
               for k in range(p))
```

18. Un chemin optimal vers (i, j) avec $i > 0$ et $j > 0$ ne peut être obtenu que de deux façons: en arrivant à (i, j) par en-dessous, ou par sa gauche.

    De plus, pour que le chemin vers (i, j) soit optimal, il faut que l'un au moins des chemins vers $(i-1, j)$ et vers $(i, j-1)$ soit optimal (si aucun des deux ne l'est, le chemin vers (i, j) peut être remplacé par un autre chemin de poids strictement inférieur, donc il ne peut pas être optimal).

    Donc la longueur d'un chemin optimal est la plus petite des deux longueurs obtenues en ajoutant $F(i-1, j)$ et $P(i-1, j, i, 1)$ d'une part pour une arrivée avec un dernier pas vers la droite, et en ajoutant $F(i, j-1)$ et $P(i, j-1, i, 1)$ d'une part pour une arrivée avec un dernier pas vers le haut.

19. On obtient

$$F(0,0) = 0,$$
$$\forall i > 0,\ F(i,0) = F(i-1,0) + P(i-1,0,i,0),$$
$$\forall j > 0,\ F(0,j) = F(0,j-1) + P(0,j-1,0,j).$$

20. La méthode récursive telle qu'elle est proposée n'est pas efficace car elle suppose de calculer de nombreuses fois les mêmes valeurs sans tenir compte du fait qu'elles ont déjà été rencontrées par le passé.

Plus précisément, le calcul de $F(i,j)$ par cette méthode nécessite de l'ordre de 2^{i+j} appels récursifs (le nombre total de chemins de $(0,0)$ à (i,j) dans le graphe), alors qu'il n'y a que $(i+1)(j+1)$ sommets à explorer au total.

21. Dans la proposition de programme suivante, chaque ligne ne fait appel qu'à des valeurs déjà calculées dans le tableau.

```
F = np.zeros(n, n)
for i in range(1, n):
    F[i, 0] = F[i-1, 0] + P[i-1, 0, i, 0]
for j in range(1, n):
    F[0, j] = F[0, j-1] + P[0, j-1, 0, j]
for i in range(1, n):
    for j in range(1, n):
        F[i, j] = min(F[i-1, j] + P[i-1, j, i, j],
                      F[i, j-1] + P[i, j-1, i, j])
print("Le poids d'un escalier optimal est ", F[n-1, n-1])
```

22. Le nombre d'additions, d'affectations et de calculs de minimum est $O(n) \cdot O(1) + O(n) \cdot O(1) + O(n) \cdot O(n) \cdot O(1) = O(n^2)$.

23. On peut par exemple procéder comme suit:

```
F = np.zeros(n, n)
D = np.zeros(n, n)
for i in range(1, n):
    F[i, 0] = F[i-1, 0] + P[i-1, 0, i, 0]
    D[i, 0] = 1
for j in range(1, n):
    F[0, j] = F[0, j-1] + P[0, j-1, 0, j]
    D[0, j] = 2
for i in range(1, n):
    for j in range(1, n):
        gauche = F[i-1, j] + P[i-1, j, i, j]
        bas    = F[i, j-1] + P[i, j-1, i, j]
        if gauche < bas:
            F[i, j] = gauche
            D[i, j] = 1
        else:
            F[i, j] = bas
            D[i, j] = 2
```

24. On remonte le chemin optimal en suivant les indications contenues dans la matrice D.

```
def reconstruire(n, D):
    i, j = n-1, n-1
    chemin = [(i, j)]
    while i > 0 or j > 0:
```

```
    if D[i, j] == 1:
        i -= 1
    else:
        j -= 1
    chemin.append((i, j))
chemin.reverse()
return chemin
```