Correction du DS 3

Julien Reichert

Partie 1

Question 1.1

Il s'agit de bottom-up, comme en témoigne la boucle croissante. On note également qu'il n'y a pas de test que la valeur est disponible, qui révèlerait un top-down.

Question 1.2

On remarque que la variable `objets` est une liste de couples (ou de listes de taille deux, par exemple), puisqu'on accède à `objets[i-1][1]` (le poids, d'après les commentaires) et à `objets[i-1][0]` (la valeur). Imaginons qu'on ait `objets = [(3, 3), (4, 5), (1, 1), (19, 8), (40, 16), (9, 6)]` et `capacite = 15`. La fonction retourne alors `[2, 3, 5]`, car la valeur maximale (de 29) associée à un poids inférieur ou égal à 15 s'obtient en prenant le troisième, le quatrième et le sixième objet (le quatrième a une valeur supérieure à tous les autres qui peuvent être pris donc il est incontournable, puis de même pour le sixième, ce qui laisse juste encore de la place pour le troisième).

Question 1.3

On peut envisager plusieurs algorithmes gloutons : un premier qui prend systématiquement l'objet encore prenable ayant le plus de valeur, un deuxième qui prend le plus d'objets possible donc par poids croissant (à poids égal, par valeur décroissante tout de même), et un troisième qui a plus de pertinence consiste à considérer les objets ordonnés par rapport valeur sur poids décroissant et de prendre à chaque fois le meilleur encore prenable.

L'instance `[(6, 5), (3, 3), (4, 4)]` avec une capacité de 7 met en échec le premier algorithme glouton ainsi que le troisième. En doublant la valeur du premier objet, c'est le deuxième algorithme qui est mis en échec.

Question 1.4

On peut faire une boucle sur le nombre d'occurrences de l'objet qu'on peut incorporer, pour une complexité qui serait multipliée par le nombre maximal d'exemplaires d'un objet qu'on peut prendre, donc a priori par la capacité.

Pour aller plus vite, on peut également se servir non pas de la réponse sans tenir compte du nouvel objet mais de la réponse en tenant compte de celui-ci (l'ordre des calculs s'y prête), en initialisant à partir de la réponse sans le nouvel objet.

Le début reste le même, et les lignes « à la frontière » sont recopiées pour bien montrer où l'on est.

```
...
            if objets[i-1][1] <= j:
                test = toutes_valeurs[i][j-objets[i-1][1]] + objets[i-1][0] # premier changement
                if toutes_valeurs[i][j] < test:
                    toutes_valeurs[i][j] = test
                    les_objets[i][j] = les_objets[i][j-objets[i-1][1]] + [i-1] # deuxième changement
    return les_objets[-1][-1]
```

Question 1.5

Les mêmes contre-exemples restent valables, le glouton n'est toujours pas optimal dans le cas général.

Question 1.6

La complexité en temps est linéaire en le produit du nombre d'objets (plus un) et de la capacité (plus un) pour les deux algorithmes (c'est aussi la complexité en espace).

Partie 2

Question 2.1

La ligne manquante triait la liste `distances` en place, par exemple `distances.sort()`.

Question 2.2

```
def dist_eucl_dim_d(pt1, pt2):
    assert len(pt1) == len(pt2) # cela ne peut pas faire de mal
    rep = 0
    for i in range(len(pt1)):
        rep += (pt1[i] - pt2[i]) ** 2
    return rep ** .5 # ou rep si on prend la distance au carré (le tri sera identique)
```

Un élément de `les_points` sera alors par exemple `((4, 2), "rouge")`, donc un point dans un espace de dimension deux, de coordonnées $(4, 2)$ et de couleur rouge.

Question 2.3

La variable `etiq` correspond à sa valeur au dernier tour de boucle, donc à l'étiquette du `k`-ième plus proche voisin (avec la gestion des éventuelles égalités faite par le tri).

Question 2.4

Changement au niveau de la boucle, avec une indentation relative :

```
for d, etiq in distances[:k]:
    if d == 0:
        return etiq
    if etiq in etiquettes:
        etiquettes[etiq] += 1 / d
    else:
        etiquettes[etiq] = 1 / d
```

Question supplémentaire

Dans le contexte du programme, il s'agirait de prendre les moyennes dans les matières présentées des étudiants et de chercher les plus proches voisins au regard des moyennes (des matières différentes garantissant un éloignement artificiel), puis d'utiliser deux étiquettes en fonction du succès.

En termes d'efficacité, le fait que le nombre de données d'apprentissage ne soit pas bien plus élevé que le nombre de données à tester est un point négatif. Par ailleurs, on peut s'attendre à ce que l'algorithme confirme simplement que la réussite est favorisée par de bonnes moyennes, limitant l'intérêt d'utiliser de l'apprentissage supervisé. Si l'algorithme ne le confirme pas, c'est éventuellement aussi que les résultats sont aléatoires.

Quoi qu'il en soit, cela pose alors de gros problèmes d'éthique, mais heureusement il n'y a aucune raison de penser que cette pratique a effectivement lieu.

Partie 3

Question 3.1

Si la case n'a pas de voisine révélée, la probabilité est simplement le nombre de bombes non encore détectées divisée par le nombre de cases non encore révélées.

Question 3.2

```
def voisines(ligne, colonne, nbl, nbc):
    rep = []
    for i in range(max(0, ligne-1), min(ligne+2, nbl)):
        for j in range(max(0, colonne-1), min(colonne+2, nbc)):
        if (i, j) != (ligne, colonne):
            rep.append((i, j))
    return rep
```

Complexité constante.

Question 3.3

```
def inconnues_et_bombes(grille, ligne, colonne):
    nbl = len(grille)
    nbc = len(grille[0])
    non_revelees = 0
    bombes = 0
    for (i, j) in voisines(ligne, colonne, nbl, nbc):
        if grille[i][j] is None:
            non_revelees += 1
        if grille[i][j] == -1:
            bombes += 1
    return non_revelees, bombes
```

Complexité constante.

Question 3.4

Si la case a au moins une voisine révélée, on ne calcule pas la probabilité selon la première question, autant ouvrir un endroit neuf si c'est plus sécurisé que dans le voisinage en question, et si ça l'est moins on économise le calcul.

D'ailleurs il faudrait stocker le résultat du calcul qui nécessite de passer sur l'ensemble de la grille dans une variable globale qui mute à chaque action pour ne pas le calculer à chaque case, et c'est d'ailleurs ce que l'on va faire avec une syntaxe un peu avancée (mais sans objets, quand même).

```
def proba_bombe_absolue(grille, nb_bombes):
    bombes_restantes = nb_bombes
    non_revelees = 0
    for une_ligne in grille:
        for element in une_ligne:
            if element is None:
                non_revelees += 1
            if element == -1:
                bombes_restantes -= 1
    return bombes_restantes / non_revelees
# Complexité linéaire en la taille de la grille,
# c'est-à-dire en le nombre de lignes fois le nombre de colonnes.
```

```
def calcule_proba_bombe_absolue(grille, nb_bombes):
    global pba
    pba = proba_bombe_absolue(grille, nb_bombes) # En particulier même complexité !

def proba_bombe(grille, ligne, colonne, nb_bombes):
    nbl = len(grille)
    nbc = len(grille[0])
    rep = None
    for (i, j) in voisines(ligne, colonne, nbl, nbc):
        if grille[i][j] is not None and grille[i][j] != -1:
            n_r, b = inconnues_et_bombes(grille, i, j)
            if rep is None or rep < n_r / (grille[i][j] - b):
                rep = n_r / (grille[i][j] - b)
    if rep is None:
        rep = pba # calcul sollicité en amont
    return rep
# Complexité constante (au carré, mais constante quand même)
```

Question 3.5

```
def case_proba_minimale(grille, nb_bombes):
    nbl = len(grille)
    nbc = len(grille[0])
    calcule_proba_bombe_absolue(grille, nb_bombes)
    rep = None
    probarep = None
    for ligne in range(nbl):
        for colonne in range(nbc):
            if grille[ligne][colonne] is None:
                p = proba_bombe(grille, ligne, colonne, nb_bombes)
                if probarep is None or probarep > p:
                    probarep = p
                    rep = (ligne, colonne)
    return rep
```

Question 3.6

Les complexités ont été calculées question par question jusqu'à la dernière exclue. Pour la dernière, grâce à l'astuce de ne solliciter le calcul qu'une seule fois pour l'ensemble des cases sans voisines non révélées, la complexité reste linéaire en la taille de la grille au lieu de quadratique (et même si aucune case n'avait pas de voisine non révélée, il fallait de toute manière parcourir la grille donc on n'y perd rien).

Partie 4

Question 4.1

Les éléments de la valeur de retour sont incorporés au fur et à mesure de la découverte du fait qu'ils sont dans l'attracteur. C'est donc par indice croissant du premier i tel que le sommet ajouté soit dans $\mathcal{A}_i$.

Question 4.2

Les fonctions `inclus` et `commun` sont maintenues, mais agissent désormais sur un dictionnaire en deuxième argument.

Pour ne pas avoir à recopier intégralement un dictionnaire dans un autre pour tenir compte des quelques changements, une autre modification de la fonction est mise en œuvre : on utilise un dictionnaire des nouveaux sommets, correspondant mathématiquement à $\mathcal{A}_{n+1} \setminus \mathcal{A}_n$.

```
def attracteur(arene, objectif):
    a_n = dict() # Ou {}
    for x in objectif:
        a_n[x] = 0
    tour = 1
    while True:
        nouveaux = dict()
        for i, (proprietaire, successeurs) in enumerate(arene):
            if inclus(successeurs, a_n) and proprietaire:
                if i not in nouveaux and successeurs != []:
                    nouveaux[i] = tour
            elif commun(successeurs, a_n) and not proprietaire:
                if i not in nouveaux:
                    nouveaux[i] = tour
        if nouveaux == {}: # ou if nouveaux ou if len(nouveaux) == 0
            return a_n
        for x in nouveaux:
            a_n[x] = nouveaux[x]
        tour += 1
```

Question 4.3

Désormais, le test d'inclusion ou d'existence d'un élément commun passe en temps linéaire en la taille du premier argument, qui est le nombre d'arcs sortant du sommet en cours d'étude.

L'ensemble des vérifications est alors linéaire en le nombre d'arcs pour tous les tours de la boucle inconditionnelle intérieure, qui est un tour de la boucle conditionnelle extérieure.

Les autres opérations sont en temps linéaire en le nombre de sommets, (recopiage des nouveaux sommets). On admettra que ceci est inférieur au nombre d'arcs.

D'après le cours, le nombre de tours de la boucle conditionnelle est majoré par le nombre de sommets.

Il vient une complexité finale de l'ordre du produit du nombre de sommets par le nombre d'arcs. En gros on a économisé un facteur n.

Question 4.4

En pratique on n'a pas besoin d'une arène mais puisque c'est la consigne...

```
def arene_renversee(arene):
    rep = [(b, []) for (b, _) in arene]
    for i, (_, successeurs) in enumerate(arene):
        for j in successeurs:
            rep[j][1].append(i)
    return rep
```

L'objectif de complexité est atteint, on ne fait que des ajouts en fin de liste sur un parcours seulement de l'arène.

Question 4.5

On renverra ici un dictionnaire mais dont les valeurs n'ont pas d'intérêt (des booléens), ce qui n'incorpore pas le travail fait à la deuxième question.

L'avantage est qu'on peut traiter les sommets de l'attracteur dans l'ordre que l'on souhaite, par exemple en se servant d'une liste comme d'une pile. Autrement, une file aurait été nécessaire et la prise en compte des valeurs dans le dictionnaire aussi.

```
def attracteur_autre_methode(arene, objectif):
    a = arene[:]
    a_rev = arene_renversee(arene)
    attracteur = dict()
    for x in objectif:
        attracteur[x] = True
    a_traiter = objectif[:]
    while len(a_traiter) > 0:
        en_cours = a_traiter.pop()
        for x in a_rev[en_cours]:
            if not a[x][0]:
                if x not in attracteur :
                    attracteur[x] = True
                    a_traiter.append(x)
            else :
                a[x][1].remove(en_cours)
                if a[x][1] == [] :
                    attracteur[x] = True
                    a_traiter.append(x)
    return(attracteur)
```

Question 4.6

Désormais, la fonction ne traite chaque sommet qu'une fois au regard de ses prédécesseurs, donc chaque arc une seule fois aussi. Reste le problème du retrait d'un arc par la fonction remove. Idéalement, la copie de l'arène est un dictionnaire dont les retraits sont en temps constant, pour un total linéaire en la taille du graphe. Ici chaque arc peut amener à faire autant de décalages que d'arcs restant, ce qui amène à multiplier la complexité annoncée précédemment par le plus grand nombre d'arcs sortant d'un sommet.

Partie 5

Question 5.1

Il suffit de créer des tables avec deux attributs, un identifiant (clé primaire) et une valeur associée (chaîne de caractères) pour toutes les chaînes de caractères apparaissant dans la base de données (sauf les dates, éventuellement, ce qui serait un peu abusif).

Les chaînes dans les tables `Spectacle` et `Distribution` seraient alors remplacées vers des clés étrangères vers les nouvelles tables.

On notera que grâce à cela, d'éventuelles homonymies dans les personnes ne poseraient pas de problème, par ailleurs.

Question 5.2

```
SELECT DISTINCT(Pays) FROM Spectacle
```

Question 5.3

```
SELECT COUNT(*) FROM Spectacle WHERE Type="Opéra"
```

Question 5.4

```
SELECT Personne FROM Distribution JOIN Spectacle ON Distribution.Id_spectacle=Spectacle.Id_spectacle
WHERE Role="Chef d'orchestre" AND Sur_scene=0 AND Titre="La Traviata"
```

Question 5.5

Requête facile pour donner un titre arbitraire :

```
SELECT Titre FROM Spectacle JOIN Distribution ON Spectacle.Id_spectacle=Distribution.Id_spectacle
GROUP BY Spectacle.Id_spectacle ORDER BY COUNT(DISTINCT Personne) DESC LIMIT 1
```

(Pas de pénalité en cas de `COUNT(*)`.)

Requête compliquée pour tous les donner :

```
SELECT Titre FROM Spectacle JOIN Distribution ON Spectacle.Id_spectacle=Distribution.Id_spectacle
GROUP BY Spectacle.Id_spectacle HAVING COUNT(DISTINCT Personne) =
(SELECT COUNT(DISTINCT Personne) FROM Spectacle JOIN Distribution
ON Spectacle.Id_spectacle=Distribution.Id_spectacle GROUP BY Spectacle.Id_spectacle
ORDER BY COUNT(DISTINCT Personne) DESC LIMIT 1)
```

Une autre version utilise une table dérivée et la fonction `MAX` pour éviter la combinaison `ORDER BY` et `LIMIT`, mais elle ne s'inspire pas de l'énoncé donc ne vient pas à l'esprit aussi bien (et l'avis général serait sans doute qu'elle est encore plus compliquée).

Question 5.6

Plutôt que de s'embêter à copier-coller, il suffit de glisser avant chaque `GROUP BY` la condition `WHERE Sur_scene`, assorti ou non d'une comparaison à `1`, identique à la comparaison d'un booléen à `True` en Python.

Question 5.7

Requête facile, la version compliquée s'obtenant par une adaptation analogue.

```
SELECT Ville FROM Spectacle GROUP BY Ville ORDER BY COUNT(DISTINCT Salle) DESC LIMIT 1
```

Remarque intéressante : j'ai pu observer que `COUNT(DISTINCT Salle)` éliminait automatiquement les cas où `Salle` valait `NULL`, ce qui est pratique car la gestion de cette valeur n'est pas au programme (et donc le barème ignorera des tentatives malheureuses).

Question 5.8

La première requête donne la liste des chefs d'orchestre (avec une vérification qu'il ne s'agit pas d'un nom de personnage, une vérification inutile dans la mesure où chaque spectacle a un chef d'orchestre et que la clé de la table exclut un doublon avec le même nom) qui ont dirigé au moins deux spectacles auxquels j'ai assisté, avec le nombre en question, celui-ci faisant office de critère de tri décroissant pour la présentation du résultat.

La deuxième requête donne la liste des artistes ayant figuré sur scène dans des spectacles auxquels j'ai assisté dans au moins deux villes différentes, ainsi que le nombre en question, avec le même tri que précédemment.

La troisième requête détermine le ou les compositeur(s) dont j'ai assisté au nombre maximal de représentations d'un opéra, ainsi que le nombre en question. Les opéras vus plusieurs fois comptent pour autant d'occurrences, et le nombre d'œuvres distinctes aurait été pris en compte si j'avais remplacé dans la requête toutes les étoiles dans les `COUNT` par `DISTINCT Titre`.