DS 3

Informatique de tronc commun, classe de PC

Julien Reichert

Ce devoir consiste en plusieurs parties indépendantes.

On rappelle la consigne universelle : une fonction qui « détermine si ... » doit renvoyer un booléen.

Sauf mention explicite du contraire, une fonction qui demande de calculer une complexité attend la complexité en temps dans le pire des cas.

Partie 1 : Programmation dynamique : le problème du sac à dos.

On donne ci-après une fonction Python résolvant le problème du sac à dos dont on rappelle l'énoncé : étant donné un ensemble d'objets d'un certain poids et d'une certaine valeur, comment remplir un sac à dos avec un exemplaire de certains objets de sorte que la valeur transportée soit maximale et le poids soit inférieur à un seuil donné ?

```
def sac_a_dos(objets, capacite):
    n = len(objets)
    toutes_valeurs = [[0 for j in range(capacite+1)] for i in range(n+1)]
    les_objets = [[[] for j in range(capacite+1)] for i in range(n+1)]
    for i in range(1, n+1):
        for j in range(1, capacite+1):
            toutes_valeurs[i][j] = toutes_valeurs[i-1][j] # Au moins aussi bien que sans l'objet i-1
            les_objets[i][j] = les_objets[i-1][j]
            if objets[i-1][1] <= j: # L'objet i-1 n'est pas trop lourd
                test = toutes_valeurs[i-1][j-objets[i-1][1]] + objets[i-1][0]
                if toutes_valeurs[i][j] < test: # En incorporant l'objet i-1, c'est mieux !
                    toutes_valeurs[i][j] = test
                    les_objets[i][j] = les_objets[i-1][j-objets[i-1][1]] + [i-1]
    return les_objets[-1][-1]
```

Question 1.1 : Le programme est-il écrit en top-down ou en bottom-up ? Justifier.

Question 1.2 : Proposer une valeur possible pour `objets` et une valeur possible pour `capacite` avec une demi-douzaine d'objets. Préciser la valeur de retour de la fonction en justifiant.

Question 1.3 : Quels algorithmes gloutons auraient intuitivement une certaine efficacité (en donner au moins un) ? Prouver par un contre-exemple (qui peut être l'instance de l'exercice précédent avec un peu de chance) que ces algorithmes gloutons ne sont pas optimaux.

En pratique, le problème du sac à dos a deux versions, dans celle qui précède, chaque objet peut être ou ne pas être pris (on appelle ceci la version 0-1), dans une autre on peut prendre autant d'exemplaires de chaque objet que l'on souhaite.

Question 1.4 : Adapter le programme précédent pour la nouvelle version du problème. Il est possible de mentionner les lignes à corriger sans recopier le reste.

Question 1.5 : Discuter de nouveau de l'optimalité des algorithmes gloutons.

Question 1.6 : Donner la complexité du programme précédent et de son adaptation.

Partie 2 : Intelligence artificielle - extension de kNN

On redonne ci-dessous une version de l'algorithme des k plus proches voisins.

```
def kNN(les_points, point_sup, k, dist):
    distances = [(dist(point[0], point_sup), point[1]) for point in les_points]
    # Il manque quelque chose ici
    etiquettes = dict()
    for _, etiq in distances[:k]:
        if etiq in etiquettes:
            etiquettes[etiq] += 1
        else:
            etiquettes[etiq] = 1
    rep = etiq # Initialisation pertinente plutôt que None
    for e in etiquettes: # Moche de réutiliser le nom etiq, mais ça marcherait
        if etiquettes[e] > etiquettes[rep]:
            rep = e
    return rep
```

Question 2.1 : La ligne consistant en un commentaire « `Il manque quelque chose` » a remplacé une ligne de code cruciale. Compléter (en une ou plusieurs lignes, en appelant éventuellement une fonction qu'on écrira / existante).

Question 2.2 : Écrire une fonction `dist` qui puisse être compatible avec la fonction `kNN`. Écrire alors un élément possible de la liste `les_points`, en expliquant la structure.

Question 2.3 : Que vaut précisément `etiq`, dans la ligne qui initialise `rep`, par rapport aux arguments de la fonction ?

Question 2.4 : On souhaite modifier la fonction de sorte que plus un des k plus proches voisins est proche, plus il va contribuer pour le choix de son étiquette. La formule suggérée est de le compter pour l'inverse de la distance au point supplémentaire, en traitant le cas particulier d'une distance nulle : dans ce cas l'étiquette sera forcément retenue (mathématiquement cohérent...). On admettra que les points fournis sont distincts deux à deux. Écrire une nouvelle version de la fonction `kNN`. On pourra signaler les modifications à faire, pour aller plus vite.

Question supplémentaire : Un professeur envisage d'utiliser l'algorithme des k plus proches voisins pour estimer les candidats qui sont plus susceptibles de réussir en les comparant aux dossiers d'étudiants passés. Discuter de la façon dont ceci pourrait être mis en place et commenter l'idée en argumentant.

Partie 3 : Minmax - Démineur

Bien qu'il soit a priori inutile d'expliquer le jeu (ne niez pas, je vois tout !), une mise en contexte ne fera pas de mal.

On considère une grille de taille `n` (lignes) par `m` (colonnes) dont `k` cases portent une bombe, ce qui sera modélisé par une matrice de booléens avec `k` occurrences de `True` exactement.

On considère également une grille de jeu de mêmes dimensions initialisée à `None`, dont les valeurs sont remplacées lors de leur révélation par le nombre de bombes dans les cases voisines latéralement et en diagonale (donc jusqu'à huit cases), sachant que la partie est perdue si la case révélée contient une bombe. Il est également possible de marquer une case par un `-1`, précisant qu'il s'agit d'une bombe identifiée (ceci ne nous concernera pas dans cet exercice).

La première révélation est sécurisée, donc la case ne contient pas de bombe. Par la suite, des déductions logiques permettent d'identifier des cases sécurisées et des cases contenant des bombes.

Cependant, il peut arriver qu'aucune déduction ne puisse être faite et qu'il faille cliquer au hasard. Dans ce cas, il peut être pertinent de cliquer sur une case qui minimise la probabilité de contenir une bombe.

Pour déterminer la probabilité de contenir une bombe d'une case `C`, il faut déjà que cette case n'ait pas encore été révélée. Pour chaque case voisine `V` révélée, donc présentant un indice, la probabilité que la case `C` contienne une bombe au regard de `V` est le nombre de bombes non encore détectées au contact de `V` divisé par le nombre de cases voisines de `V` non encore révélées. On considère alors la probabilité que `C` contienne une bombe comme le maximum des probabilités au regard des voisines révélées.

[En pratique, le lien avec minmax n'est pas le plus pertinent, mais bon...]

Question 3.1 : Quelle est la formule la plus pertinente pour calculer la probabilité si une case n'a pas de voisine révélée ?

Question 3.2 : Écrire une fonction permettant d'obtenir la liste des voisines d'une case en tant que couples de coordonnées (ligne, colonne), en fonction de la case actuelle et des dimensions de la grille.

Question 3.3 : Écrire une fonction permettant d'obtenir le nombre de cases non révélées et de bombes détectées dans le voisinage d'une case, en fonction de la case considérée et de la grille de jeu.

Question 3.4 : Écrire une fonction permettant d'obtenir la probabilité qu'une case contienne une bombe, en fonction de la case considérée, de la grille de jeu et du nombre total de bombes.

Question 3.5 : Écrire une fonction permettant de déterminer une case minimisant cette probabilité de contenir une bombe, en fonction de la grille de jeu et du nombre total de bombes.

Question 3.6 : Déterminer la complexité de toutes les fonctions écrites.

Partie 4 : Théorie des jeux

On donne ici une fonction pour calculer l'attracteur dans le cas d'une condition d'accessibilité pour Ève avec les notations du TP 6, c'est-à-dire que les arènes sont représentées en Python comme des listes dont les éléments, simulant les sommets, sont des couples dont le premier élément est le propriétaire du sommet (booléen valant `False` pour Ève et `True` pour Adam) et le deuxième est la liste des indices où figurent les couples représentant les sommets auxquels le sommet est relié.

```
def inclus(liste1, liste2):
    for element in liste1:
        if element not in liste2:
            return False
    return True

def commun(liste1, liste2):
    for element in liste1:
        if element in liste2:
            return True
    return False

def attracteur(arene, objectif):
    a_n = []
    a_np1 = objectif
    while a_np1 != a_n:
        a_n = a_np1[:]
        for i, (proprietaire, successeurs) in enumerate(arene):
            if inclus(successeurs, a_n) and proprietaire:
                if i not in a_np1 and successeurs != []:
                    a_np1.append(i)
            elif commun(successeurs, a_n) and not proprietaire:
                if i not in a_np1:
                    a_np1.append(i)
    return a_n
```

Question 4.1 : Dans quel ordre figurent les éléments de la valeur de retour ?

Il s'avère que la complexité (nombre de sommets à la puissance quatre ou, en raffinant, nombre de sommets au carré fois nombre d'arcs) n'est pas très bonne, elle a été sacrifiée au prix de la lisibilité. L'essentiel de cet exercice consistera à l'améliorer. Mais pour commencer, une adaptation du programme est attendue.

Question 4.2 : Adapter le code pour que `a_n` soit un dictionnaire dont les clés seront les sommets dans $\mathcal{A}$ et les valeurs seront pour chaque sommet $s \in \mathcal{A}$ le plus petit indice k tel que s soit dans $\mathcal{A}_k$.

Pour l'exercice précédent, il est possible de tout réécrire depuis zéro ou de reprendre le programme en signalant les modifications apportées.

Question 4.3 : Quelle est la nouvelle complexité en utilisant des dictionnaires (on notera que la valeur associée à chaque clé n'a pas d'impact sur la complexité) ? On admettra que toute opération sur le dictionnaire est en temps constant.

Pour améliorer encore la complexité et ne considérer chaque arc qu'une fois, nous allons utiliser le principe précédent : le calcul de l'attracteur se fera en suivant les arcs à l'envers. Si un arc arrivant dans un sommet de l'attracteur part d'un sommet d'Ève, ce sommet est déclaré gagnant (s'il ne l'était pas encore), si un arc arrivant dans un sommet de l'attracteur part d'un sommet d'Adam, cet arc est retiré (on travaille sur une copie de l'arène pour ne pas détruire l'argument), et si le sommet d'Adam n'a plus d'arc sortant de ce fait, il est ajouté à l'attracteur. On notera qu'un sommet d'Adam sans arc sortant ne sera alors jamais considéré, ce qui permet de ne pas avoir de risque de faux positif.

Question 4.4 : Écrire une fonction prenant en argument une arène et qui construit l'arène dans laquelle les arcs sont renversés. Il faut que la complexité soit linéaire en la taille du graphe (donc la somme du nombre de sommets et du nombre d'arcs). Si ce n'est pas le cas, il y aura une pénalité, mais le calcul final de complexité considèrera que c'est le cas.

Question 4.5 : Écrire fonction calculant l'attracteur en tenant compte des instructions ci-avant. La valeur de retour peut être une liste ou un dictionnaire, au choix.

Question 4.6 : Quelle est la complexité de la fonction écrite ici ?

Partie 5 : Bases de données

Pour cet exercice, nous utiliserons une version simplifiée d'une base de données que j'ai créée pendant le confinement pour recenser les spectacles musicaux auxquels j'ai assisté, ainsi que diverses informations à leur propos.

- Table Spectacle, avec les attributs Id_spectacle (clé primaire avec auto-incrémentation), Date (chaîne de caractères [1]), Titre (chaîne de caractères), Compositeur (chaîne de caractères), Type (chaîne de caractères), Pays (chaîne de caractères), Ville (chaîne de caractères) et Salle (chaîne de caractères). Le couple (Date, Titre) est aussi une clé, mais pas l'attribut Date [2].
- Table Distribution, avec les attributs Id_spectacle (clé étrangère pour la table Spectacle), Sur_scene (booléen, assimilable à un entier valant forcément 0 ou 1, respectivement pour faux et vrai), Role (chaîne de caractères) et Personne (chaîne de caractères). Le couple (Id_spectacle, Role) est une clé.

Pour clarifier, les valeurs possibles de l'attribut Type de la table Spectacle sont `"Opéra"`, `"Ballet"`, `"Comédie musicale"` et `"Concert"`, ce qui permet de ne pas se tromper quand ces valeurs apparaitront dans une question. De plus, l'attribut Sur_scene est à faux pour des valeurs de Role telles que `"Chef d'orchestre"`, `"Metteur en scène"`, etc. On en déduit que Personne est l'attribut correspondant à l'identité des chanteurs / acteurs / directeurs...

Question 5.1 : Proposer une structure étendue pour cette base de données afin d'éviter d'y faire figurer une chaîne de caractères deux fois. Tous les attributs des nouvelles tables sont à expliquer en détaillant les clés primaires et étrangères. Ceci étant, **seule la structure de l'énoncé servira pour les requêtes par la suite**.

Question 5.2 : Écrire une requête permettant d'obtenir la liste des pays où j'ai assisté à au moins un spectacle.

Question 5.3 : Écrire une requête permettant d'obtenir le nombre d'opéras auxquels j'ai assisté.

Question 5.4 : Écrire une requête permettant d'obtenir le nom des chefs d'orchestre des productions de "La Traviata" que j'ai vues. [3]

Question 5.5 : Écrire une requête permettant d'obtenir le titre du spectacle avec le plus de personnes enregistrés dans la base de données au niveau de la distribution. En cas d'égalité, on pourra au choix renseigner un titre arbitraire ou tous.

1. ... au format JJ/MM/AAAA, pas très idéal pour trier par ordre chronologique, heureusement qu'il y a l'identifiant pour cela !
2. Je peux être très motivé certains jours...
3. Pas besoin de gérer d'éventuels doublons, il n'y en a d'ailleurs pas à l'heure actuelle.

Question 5.6 : Même question en se restreignant aux rôles sur scène.

Question 5.7 : Écrire une requête permettant d'obtenir le nom de la ville où il y a eu le plus grand nombre de salles différentes dans lesquelles j'ai assisté à au moins un spectacle. Même consigne quant aux égalités éventuelles. [4]

Question 5.8 : Expliquer ce que font les trois requêtes ci-dessous.

```
SELECT Personne, COUNT(*)
FROM Spectacle JOIN Distribution
ON Spectacle.Id_spectacle=Distribution.Id_spectacle
WHERE Role="Chef d'orchestre" AND Sur_scene=0
GROUP BY Personne
HAVING COUNT(*) > 1
ORDER BY COUNT(*) DESC
```

```
SELECT Personne, COUNT(DISTINCT Ville)
FROM Spectacle JOIN Distribution
ON Spectacle.Id_spectacle=Distribution.Id_spectacle
WHERE Sur_scene=1
GROUP BY Personne
HAVING COUNT(DISTINCT Ville) > 1
ORDER BY COUNT(DISTINCT Ville) DESC
```

```
SELECT Compositeur, COUNT(*)
FROM Spectacle
WHERE Type="Opéra"
GROUP BY Compositeur
HAVING COUNT(*) =
(
  SELECT COUNT(*)
  FROM Spectacle
  WHERE Type="Opéra"
  GROUP BY Compositeur
  ORDER BY COUNT(*) DESC
  LIMIT 1
)
```

4. La réponse SELECT "Paris" n'est pas autorisée !