Informatique PCSI
Prérequis TP 6 : algorithmes gloutons

1 Problème du sac à dos

Nous considérons la variante « entière » du problème du sac à dos.

Nous sommes devant un ensemble de n objets. Chaque objet noté o_i a une valeur notée v_i et un poids noté p_i. Il s'agit d'emporter dans son sac à dos l'ensemble d'objets qui a la plus grande valeur sachant que le sac supporte un poids maximum P. Comment résoudre ce problème, quels objets doit-on prendre ?

Pour appliquer une stratégie gloutonne, nous devons définir ce que nous entendons par le meilleur choix à chaque étape.

Il y a trois manières ici de définir un meilleur choix. Parmi les objets qui n'ont pas encore été pris, soit on choisit un objet qui a la valeur maximale, soit un objet qui a le poids minimal, soit un objet qui a le rapport valeur/poids maximal.

Nous ne considérons que les objets ayant un poids $p_i \leq P$. L'algorithme glouton consiste, à chaque étape, à choisir parmi ces objets celui qui représente le choix optimal. Nous le notons $O1$, sa valeur $V1$ et son poids $P1$. Ensuite, nous recommençons parmi les objets de poids $p_i \leq P - P1$. Et ainsi de suite.

Cette variante est dite entière parce que chaque objet est pris ou pas. Il existe une version fractionnaire. Le principe de base est le même, mais cette fois il est possible de prendre des fractions d'objets. Un algorithme glouton donne une solution optimale à cette variante fractionnaire.

Prenons un exemple : le sac à dos peut contenir 15 Kgs. Les poids des objets sont en Kg, les valeurs en euro.

Objet	Valeur	Poids	Valeur/Poids
Objet 1	126	14	9
Objet 2	32	2	16
Objet 3	20	5	4
Objet 4	5	1	5
Objet 5	18	6	3
Objet 6	80	8	10

Un objet est représenté par une liste comme `['objet 1', 126, 14]`.

Nous définissons trois fonctions qui prennent en paramètre un objet et renvoient respectivement la valeur, l'inverse du poids et le rapport valeur/poids de l'objet.

Remarque : si le poids est minimal, son inverse est maximal.

```
def valeur(obj):
    return obj[1]

def poids(obj):
    return 1 / obj[2]

def rapport(obj):
    return obj[1] / obj[2]
```

Nous définissons ensuite une fonction `glouton` qui prend en paramètres une liste d'objets, un poids maximal (celui que peut supporter le sac à dos) et le type de choix utilisé (par valeur, par poids, ou par

valeur/poids). La première chose à faire est de trier la liste suivant le type de choix utilisé par ordre décroissant. Nous utilisons la fonction `sorted`. Une variable réponse sert à stocker les objets choisis.

Ensuite nous parcourons la liste triée et ajoutons les noms des objets un par un tant que le poids total ne dépasse pas le poids maximal. La valeur totale et le poids total sont stockés dans deux variables `valeur` et `poids`.

```
def glouton(liste, poids_max, choix):
    copie = sorted(liste, key=choix, reverse=True)
    reponse = []
    valeur = 0
    poids = 0
    i = 0
    while i < len(liste) and poids <= poids_max:
        nom, val, pds = copie[i]
        if poids + pds <= poids_max:
            reponse.append(nom)
            poids = poids + pds
            valeur = valeur + val
        i = i + 1
    return reponse, valeur
```

Il reste à définir la liste d'objets puis à exécuter la fonction `glouton` en précisant le type de choix utilisé.

```
objets = [['objet 1', 126, 14], ['objet 2', 32, 2],
          ['objet 3', 20, 5], ['objet 4', 5, 1],
          ['objet 5', 18, 6], ['objet 6', 80, 8]]

print(glouton(objets, 15, valeur))
print(glouton(objets, 15, poids))
print(glouton(objets, 15, rapport))
```

Nous exécutons le programme et obtenons des résultats différents suivant le type de choix.

Par valeur : `(['objet 1', 'objet 4'], 131)`.

Par poids : `(['objet 4', 'objet 2', 'objet 3', 'objet 5'], 75)`.

Par valeur/poids : `(['objet 2', 'objet 6', 'objet 4'], 117)`.

Le critère « valeur » est le plus intéressant, puisque la valeur totale est 131. Mais est-ce une solution optimale ?

La réponse à cette question est non et une étude exhaustive nous montre que la solution optimale est `(['objet 2', 'objet 3', 'objet 6'], 132)`.

Dans une version fractionnaire du problème où l'on peut choisir une fraction du poids maximal autorisé pour chaque objet (par exemple s'il s'agit de poudre ou de liquide), le meilleur choix est dicté par le critère valeur/poids. Une stratégie gloutonne ferait donc prendre en premier les 2 kilos d'objet 2, puis les 8 kilos d'objet 6 et 5 kilos d'objet 1. Dans ce cas, nous obtenons finalement une valeur totale de $32 + 80 + 45 = 157$.

2 Stations d'essence

Voici un problème pour lequel un algorithme glouton donne une solution optimale.

Un automobiliste part en vacances et doit parcourir un long trajet. Il prend la route avec le plein de carburant. Son véhicule peut parcourir une distance maximale d avec un plein. La route empruntée comporte n stations services, $s_0, s_1, \ldots, s_{n-1}$, rangées dans l'ordre rencontré en suivant le parcours. La première est à une distance d_0 du départ, la deuxième est à une distance d_1 de la première, et ainsi de suite. Le point d'arrivée est à une distance d_n de la dernière station.

Une condition nécessaire et suffisante pour que l'automobiliste puisse effectuer le parcours est que la distance entre le point de départ et la première station, les distances entre deux stations consécutives et la distance entre la dernière station et le point d'arrivée soient toutes inférieures ou égales à d. Nous supposons que cette condition est remplie.

L'objectif de l'automobiliste est de s'arrêter pour faire le plein le moins souvent possible.

Un algorithme glouton consiste chaque fois que le plein est fait à parcourir le plus long trajet possible sans refaire le plein. Le plein est donc fait chaque fois à la dernière station avant de tomber en panne.

Le programme consiste en une fonction prenant deux paramètres : une liste de distances et la distance maximale qui peut être parcourue avec un plein. Les éléments de la liste sont la distance entre le point de départ et la première station, puis les distances entre deux stations consécutives et enfin la distance entre la dernière station et le point d'arrivée.

```
def glouton(distances, dmax):
    n = len(distances)
    d = dmax
    stations = []
    i = 0
    while i != n:
        while i < n and distances[i] <= d:
            d = d - distances[i]
            i = i + 1
        stations.append(i-1)
        d = dmax
    return stations
```

Pour tester le programme, on construit une liste de distances au hasard en modifiant la dernière valeur pour que le total des distances soit égal à la longueur du trajet. On précise la longueur totale du trajet et la distance qui peut être parcourue avec un plein.

```
from random import randint
trajet = 2300 # km
tab = [randint(25, 100)] # distance première station
while sum(tab) < trajet:
    tab.append(randint(25, 100)) # distances inter stations
# calcul de la dernière distance
tab[len(tab)-1] = trajet - (sum(tab) - tab[len(tab)-1])
res = 600 # 600 km avec le plein du réservoir

print(glouton(tab, res))
```

Sur un test, on obtient : `[7, 17, 26, 35]`, ce qui correspond à trois arrêts pour faire le plein aux stations d'indices 7, 17, 26. La dernière valeur est le point d'arrivée.

Les distances parcourues sont :

```
>>> sum(tab[:8])
590
>>> sum(tab[8:18])
594
>>> sum(tab[18:27])
560
>>> sum(tab[27:])
556
```

Nous allons démontrer que la stratégie utilisée est optimale.

Pour cela, considérons une solution $s = [s_1, s_2, \dots, s_p]$ obtenue avec notre algorithme et une solution optimale $S = [S_1, S_2, \dots, S_q]$.

Les deux solutions représentent des suites de stations triées de celle la plus proche du point de départ à celle la plus éloignée.

Si S est optimale, alors $q \leq p$ et nous devons prouver que $p = q$.

Soit k le plus petit entier tel que $s_k \neq S_k$. Par définition de l'algorithme glouton, $s_k > S_k$ et de plus la solution S peut s'écrire $S = [s_1, s_2, \dots s_{k-1}, S_k, S_{k+1}, \dots, S_q]$.

Montrons alors que la suite $S' = [s_1, s_2, \dots s_{k-1}, s_k, S_{k+1}, \dots, S_q]$ est elle aussi une solution optimale.

S' et S ont la même taille k, donc il suffit de vérifier que S' est une solution, c'est-à-dire qu'il n'y a aucun risque de panne. Or, puisque s est une solution, $s_k - s_{k-1} \leq d$. (d est la distance maximale qui peut être parcourue avec un plein). S est aussi une solution, donc $S_{k+1} - S_k \leq d$, d'où $S_{k+1} - s_k < S_{k+1} - S_k \leq d$, car $s_k > S_k$.

Remarquons que $S_{k+1} < s_k$, signifie qu'on pourrait supprimer S_{k+1} de S' et dans ce cas, on obtiendrait une meilleure solution, ce qui est impossible.

On itère ensuite ce raisonnement jusqu'à obtenir une solution $s = [s_1, s_2, \dots, s_q]$ qui est une solution optimale. D'où $p = q$, et la démonstration est terminée.