Chapitre 10 : Algorithmes gloutons

1. Une question de choix

Un problème d'optimisation se caractérise par une fonction que l'on cherche à maximiser ou à minimiser tout en respectant une série de contraintes auxquelles il faut satisfaire.

Il existe alors plusieurs façons d'apporter une solution à un tel problème :
- On peut imaginer un algorithme exhaustif (traitant l'ensemble du problème) qui parmi toutes les possibilités retient celle qui est globalement la meilleure. On parle alors de choix globalement optimal. Le soucis d'un tel algorithme est qu'il aura un coût beaucoup trop important et sera donc en pratique inutilisable.
- On peut au contraire s'orienter vers un algorithme qui, à chaque étape où se présentera un choix à effectuer parmi un ensemble restreint de propositions, retiendra la meilleure ; on parle alors de choix localement optimal. Un tel algorithme est appelé algorithme glouton.

La question est de savoir si une série de choix localement optimaux permet d'aboutir à une solution globalement optimale. Cela peut être le cas mais nous verrons que ce n'est pas systématique.

2. Un premier exemple : problème de rendu de monnaie

2.1.Présentation du problème

Soit r, une somme correspondant à un nombre entier, à restituer à l'aide de pièces de monnaie de différentes valeurs. On choisit de noter $p_0, \dots, p_n$, par exemple pour la monnaie européenne $(1;2;5;10;20;50;100;200;500)$ où 100 et 200 représentent les pièces de $1€$ et $2€$ et 500 le billet de $5€$, les différentes valeurs possibles rangées par ordre croissant.

Le problème du rendu monnaie consiste à trouver une série de nombres entiers $(x_0,...,x_n)$ tels que l'on ait :

$$x_0 p_0 + ... + x_n p_n = r$$

Tout en cherchant à minimiser le nombre pièces rendues. La relation ci-dessus constitue la contrainte à satisfaire. Il est à noter que si $p_0 > 1$, il ne sera pas toujours possible de rendre la somme exacte.

2.2.Étude d'un cas concret - Premier algorithme

Considérons que la somme à rendre est égale à 8 centimes d'euro. On peut remarquer sur cet exemple simple qu'il y a plusieurs possibilités mais que celles-ci s'appuient toutes sur les pièces de valeur 1, 2 et 5 donc une solution du type $[x_0,x_1,x_2,0,0,0,0,0,0]$ qu'on notera de manière abrégée $[x_0,x_1,x_2]$.

On peut par exemple :
- rendre 4 pièces de 2 centimes ($[0;4;0]$)
- huit pièces de 1 centimes ($[8;0;0]$)
- trois pièces de 1, 2 et 5 centimes qui correspond à la solution optimale.

L'algorithme suivant permet de trouver l'ensemble des solutions à ce problème précis du rendu de 8 centimes et affiche la solution optimale.

```
p=[1,2,5]
nb=8                              # initialisation du nombre de pièces
for i in range (9):               # on a au maximum 8 pièces de 1
      for j in range(5):          # on a au maximum 4 pièces de 2
            for k in range(2):    # on a au maximum 1 pièce de 5
                  s=i*p[0]+j*p[1]+k*p[2]
                  if s==8 and i+j+k<nb:
                        nb=i+j+k
                        res=[i,j,k]
print(res)
```

Même si cet algorithme fonctionne et fournit le résultat attendu, il est à noter qu'il ne peut convenir qu'à ce problème précis. Par ailleurs, ce sont trois boucles non-conditionnelles imbriquées (une par type de pièce) qui sont utilisées. En insérant un compteur, on constate qu'au total, ce sont 90 itérations qui ont été réalisées.

```
p=[1,2,5]
nb=8                              # initialisation du nombre de pièces
cpt=0                             # initialisation du compteur
for i in range (9):               # on a au maximum 8 pièces de 1
      for j in range(5):          # on a au maximum 4 pièces de 2
            for k in range(2):    # on a au maximum 1 pièce de 5
                  s=i*p[0]+j*p[1]+k*p[2]
                  cpt+=1
                  if s==8 and i+j+k<nb:
                        nb=i+j+k
                        res=[i,j,k]
print(res,cpt)
```

En se limitant aux pièces de 1, 2 et 5 centimes, la somme rendue ne peut excéder 9 (au-delà, on commence à utiliser des pièces de 10). Le nombre maximale d'itérations est obtenu pour une somme égale à 9 ; on a alors :

$$\frac{10}{1}\times\frac{10}{2}\times\frac{10}{5}=100 \text{ itérations}$$

Si on souhaite étendre la recherche des sommes supérieures, il faudra alors :

- Pour $r\leq 19$, ajouter les pièces de 10 centimes et donc une boucle supplémentaire soit au final jusqu'à 1600 itérations possibles
- ...
- Pour $r\leq 999$ (en supposant qu'à partir de 1000, on pourra utiliser des billets de $10€$), ajouter les pièces de 10, 20, 50, 100, 200 et 500 centimes et donc six boucles supplémentaires soit jusqu'à :
$$\frac{1000}{1}\times\frac{1000}{2}\times\frac{1000}{5}\times\frac{1000}{10}\times\frac{1000}{20}\times\frac{1000}{50}\times\frac{1000}{100}\times\frac{1000}{200}\times\frac{1000}{500}=10^{15} \text{ itérations}$$

Un tel algorithme n'est donc pas adapté car il peut conduire à des temps d'exécution très importants. Le problème majeur est que toutes les combinaisons possibles de pièces sont testées y compris celles ne correspondant pas à la somme à rendre.

2.3.Algorithme glouton du rendu de monnaie

On peut concevoir un algorithme plus performant en s'appuyant sur le constat suivant : pour une somme r donnée, la solution optimale est celle pour laquelle on a utilisé un maximum de pièces dont la valeur v est immédiatement inférieure ou égale à la somme à rendre.
On peut alors mettre en place le protocole suivant :

- À la somme r à rendre, on retire autant de fois que possible la valeur v de la pièce de plus forte valeur telle que $v \leq r$. On effectue cette opération jusqu'à ce que le reste r' soit tel que $r' < v$.

Exemple : $r = 47$; on a $v = 20$ soit $r' = 47 - 20 = 27 > v$ puis $r' = 27 - 20 = 7 < v$

- On recommence pour r' le même processus que pour r mais avec une nouvelle valeur de pièce v'.

Exemple : $r'' = 7 - 5 = 2 < v'$

- Le processus lorsqu'on obtient un reste nul.

Exemple : $r'' = 2$ et $v'' = 2$; $r''' = 2 - 2 = 0$. La solution est donc $[0\,;1\,;1\,;0\,;2\,;0\,;0\,;0\,;0]$.

Pour chaque situation, on retient la solution qui constitue un optimum local. Il s'agit donc d'un algorithme glouton.

Écrire une fonction `monnaie` permettant d'établir la combinaison de pièces appartement au système monétaire `p` correspondant à la somme `r` à restituer, `p` et `r` étant passés en arguments.

Dans le cas général, avec un système différent de celui montré en exemple, c'est un problème difficile à résoudre. Mais, dans presque tous les systèmes de monnaie, l'algorithme glouton est optimal. Pour rendre la monnaie, on rend la pièce (ou le billet) de valeur maximale, et on continue tant qu'il reste quelque chose à rendre.

Il arrive pourtant parfois qu'une série de choix localement optimaux ne conduise pas à un résultat globalement optimal. Illustrons cela par un exemple : le système monétaire utilisé en Angleterre jusque dans les années 1960 comportait une multitude de pièces : des pièces de 1 penny, 3 pence, 4 pence, 6 pence, 12 pence soit 1 shilling, etc... Pour rendre 8 pence par exemple, l'algorithme glouton donne une pièce de 6 pence et deux de 1 penny alors que le choix optimal est de deux pièces de 4 pence.

3. Un deuxième exemple : problème du sac à dos

3.1. Présentation du problème

Soit un ensemble de n objets notés o_i, de valeurs respectives v_i et de poids p_i. L'objectif est d'emporter dans son sac à dos la collection d'objets ayant la plus grande valeur sachant que le sac peut supporter un poids maximal P.

Pour simplifier, nous allons considérer une version « entière » de ce problème c'est-à-dire que les objets ne peuvent être divisés ; on prend l'objet dans le sac ou on le laisse. Il existe une version « fractionnaire » de ce problème dans laquelle il est possible de prendre des fractions d'objet.

Un premier algorithme consisterait à envisager toutes les combinaisons d'objets pour lesquelles le poids total est inférieur à P puis à rechercher celle ayant la valeur la plus grande. Toutefois, il parait évident que cet algorithme, pourtant déjà assez complexe, entraînera des temps d'exécution beaucoup trop grands.

3.2. Stratégie gloutonne

Pour élaborer une stratégie gloutonne, nous devons commencer par définir ce qui correspondra à un choix localement optimal.

Considérons que nous avons déjà un certain nombre d'objets dans notre sac. Parmi les objets restants, comment choisir le prochain ? Trois stratégies peuvent alors être envisagées :

a. On prend l'objet ayant le poids le plus faible
b. On prend l'objet ayant la valeur maximale
c. On prend l'objet ayant le rapport valeur/poids le plus élevé

Appuyons-nous sur l'exemple concret suivant pour un sac supportant un poids maximal de $15\,kg$. Les objets possibles sont répertoriés dans le tableau suivant, les poids étant donné en kg et les valeurs en €.

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

Il conviendra par la suite d'organiser les objets dans le tableau suivant le critère de choix (a, b, ou c) retenu et, à chaque choix d'objet qui se présente, sélectionner celui qui est en tête de liste.

3.3.Algorithmes

Pour plus de simplicité, chaque objet sera représenté par une liste contenant son nom (chaîne de caractère), sa valeur et son poids (tous deux entiers naturels).
Exemple : `['objet 1',126,14]`

On définit trois fonctions renvoyant respectivement la valeur, l'inverse du poids et le rapport valeur/poids pour un objet passé en argument.

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

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

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

En considérant l'inverse du poids et non le poids lui-même, on s'assure que le choix repose sur le fait qu'une grandeur est maximale. En effet, l'objet de poids le plus faible correspond à l'objet dont l'inverse du poids est le plus important.

Afin de trier par ordre décroissant la liste d'objet suivant le critère de choix retenu, nous utiliserons la fonction `sorted`. L'algorithme ci-dessous définit une fonction glouton qui prend en paramètres une liste d'objets, un poids maximal (celui que peut supporter le sac) et le type de choix utilisé (par valeur, par poids ou par valeur/poids).

```
def glouton(liste,poids_max,choix):
      copie=sorted(liste, key=choix, reverse=True) # Tri de la liste
      reponse=[] # Liste des objets retenues
      valeur=0   # Valeur totale des objets contenus dans le sac
      poids=0    # Poids du sac
      i=0        # Nombre d'objets contenus dans le sac
      while i<len(liste) and poids<poids_max:
            nom,val,pds=copie[i]
            if poids+pds<=poids_max:
                  réponse.append(nom)
                  poids=poids+pds
                  valeur=valeur+val
            i=i+1
      return reponse,valeur
```

On définit la liste d'objets disponibles :
`objets=[['objet 1',126,14],['objet 2',32,2],['objet3',20,5],['objet 4',5,1],['objet 5',18,6],['objet 6',80,8]]`

On peut alors exécuter le programme et on constate qu'on obtient des résultats différents suivant le critère de choix retenu. On note que le critère valeur est celui qui donne le résultat le plus intéressant puisque la valeur du contenu du sac est la plus élevée des trois propositions. Toutefois, est-ce la solution optimale ?

Si l'on effectue une recherche exhaustive, on se rend compte que la meilleure solution est la suivante :

```
(['objet 2', 'objet 3', 'objet 6'],132)
```

L'algorithme glouton ne permet donc pas dans ce cas d'obtenir une solution globalement optimale.

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 d'une poudre ou d'un liquide), le meilleur choix est dicté par le critère valeur/poids. Une stratégie gloutonne ferait donc prendre en premier les $2\,kg$ d'objet 2, puis les $8\,kg$ d'objet 6 puis $5\,kg$ d'objet 1. Dans ce cas, nous obtenons finalement une valeur totale $32+80+45=157$.