Tutoriel : Les algorithmes gloutons

Table des matières

Les algorithmes gloutons

Bonjour à tous !

Aujourd'hui nous allons parler des algorithmes gloutons qui sont des algorithmes couramment utilisés dans la résolution de problèmes.

Le principe de tels algorithmes consiste à choisir des solutions locales optimales d'un problème dans le but d'obtenir une solution optimale globale au problème.

Je vous propose de découvrir cette méthode au travers de quelques exemples.

Le problème du rendu de monnaie

Problème

Ce problème est un grand classique de l'algorithmique.
Lorsque vous passez à la caisse d'un magasin quelconque, il n'est pas rare que le caissier doive vous rendre de l'argent car le montant que vous lui avez donné est supérieur à celui que vous devez payer.

Supposons qu'on doive vous rendre la somme de 2,63€.
Il y a plusieurs façons de procéder. On peut par exemple vous rendre 263 pièces de 1 cent, 125 pièces de 2 cents et 13 pièces de 1 cent ou encore 5 pièces de 50 cents, une de 10 cents, une de 2 cents et enfin une de 1 cent.

Il y a bien évidemment énormément de possibilités pour vous rendre la dite somme.
Il y a fort à parier que les solutions du type "263 pièces de 1 cent" ne vous conviennent pas, pour cause, personne n'a envie de remplir son porte monnaie avec autant de pièces...

Le problème qui se pose est donc de minimiser le nombre de pièces rendues pour un montant fixé.

Solution naïve

La solution a laquelle on pense immédiatement est d'énumérer toutes les combinaisons de possibles, de sélectionner celles qui impliquent un minimum de pièces et de choisir la meilleure.

Cette solution, dite de force brute, fonctionnera toujours mais est très loin d'être efficace.
En effet, si elle est simple dans certains cas, elle implique en général un nombre très important de combinaisons différentes, ce qui nuit grandement à l'efficacité de notre solution.

Notre tâche va donc être de formuler une solution plus efficace pour ce type de problème.

Méthode

La méthode gloutonne vise donc à optimiser la résolution d'un problème en partant du principe suivant : des choix locaux optimaux, étape après étape, devraient produire un résultat global optimal.

Dans notre cas, on va répéter le choix de la pièce de plus grande valeur qui ne dépasse pas la somme restante.
Prenons un exemple concret, celui que nous avons introduit précédemment.
On doit donc nous rendre la somme de 2,63€ et on dispose du système de monnaie européen, à savoir ceci :

```
PIÈCES (en cents) : [1,2,5,10,20,50,100,200]
```

Appliquons donc la méthode gloutonne pour voir le choix à faire dans ce cas ci.

```
ÉTAPE 1 :
 - Somme à rendre : 263 cents.
 - Solution locale : 200.
 - Pièces utilisées : 1*2€.

ÉTAPE 2 :
 - Somme à rendre : 63 cents.
 - Solution locale : 50.
 - Pièces utilisées : 1*2€ + 1*50cents.

ÉTAPE 3 :
 - Somme à rendre : 13 cents.
 - Solution locale : 10.
 - Pièces utilisées : 1*2€ + 1*50cents +1*10cents.

ÉTAPE 4 :
 - Somme à rendre : 3 cents.
 - Solution locale : 2.
 - Pièces utilisées : 1*2€ + 1*50cents +1*10cents +1*2cents

ÉTAPE 5 :
 - Somme à rendre : 1 cent
 - Solution locale : 1
 - Pièces utilisées : 1*2€ + 1*50cents +1*10cents +1*2cents +1*1cent

On a rendu toute la monnaie, on s'arrête là !
```

Le principe est donc extrêmement simple et conduit à une solution optimale dans ce cas-ci. Nous verrons par après que ce n'est pas toujours le cas et nous verrons aussi pourquoi.

Implémentation

L'implémentation de cette solution est relativement intuitive.

On va récupérer les données sous forme d'une liste (pour le système monétaire en vigueur) et d'un entier (pour le rendu).
De là, tant que le rendu est supérieur à la pièce de plus haute valeur (située en première position dans la liste) on retranchera la valeur de cette pièce au rendu et on ajoutera la pièce dans une liste qui constituera la solution.

Si le rendu est inférieur à la pièce de plus haute valeur en cours, la fonction s'appellera récursivement en ne considérant plus la pièce de plus haute valeur. C'est alors la seconde pièce qui joue ce rôle, et ainsi de suite.

```
def greedyMethod(moneySystem, giveBack, solution):
        if giveBack == 0:
                return solution
        while giveBack >= moneySystem[0]:
                giveBack -= moneySystem[0]
                solution.append(moneySystem[0])
        else:
                return greedyMethod(moneySystem[1:len(moneySystem)], giveBack, solution)

def retrievingData():
        moneySystem = input("Entrez le système monétaire de façon décroissante sous forme d
e liste (e.g. [200,100,50,20,10,5,2,1] pour le système européen)")
        giveBack = input("Entrez la somme à rendre")
        solution = []
        if moneySystem == [] or giveBack <= 0:
                print "Vous vous êtes trompé lors de l'encodage des données"
        else:
                return greedyMethod(moneySystem, giveBack, solution)

print "Le choix proposé par la méthode gloutonne est :", retrievingData()
```

La terminaison de la récursion est garantie puisque à chaque instance le rendu ou le nombre de pièce disponible diminuent.

The 0-1 knapsack problem

Problème

On dispose d'un set S contenant n objets. Chaque objet i possède une valeur b_{i} et un poids w_{i}.

On souhaiterait prendre une partie T de ces objets dans notre sac-à-dos, malheureusement, ce dernier dispose d'une capacité limitée W. On ne pourra pas toujours mettre tous les objets dans le sac étant donné que la somme des poids des objets ne peut pas dépasser la capacité maximale.
On va cependant chercher à maximiser la somme des valeurs des objets qu'on va emporter avec soi.

Mathématiquement, cela se traduit par \max_{T\subseteq S}\sum_{i\in T}b_{i}\mbox{ avec }\sum_{i \in T} w_{i}\leq W.

Solution naïve

On pourrait être tenté d'énumérer toutes les combinaisons d'objets possibles qui satisfont à la capacité maximale du sac ou qui s'en rapprochent (le sac ne doit pas être obligatoirement rempli à fond). Néanmoins, on arrive rapidement à des calculs lourds, rendant le programme inefficace.

À nouveau, la solution de force brute fonctionne, mais ne doit pas être choisie.
Comme vous vous en doutez, on va résoudre ce problème au moyen de la méthode gloutonne.

Méthode

L'idée à suivre, si on veut développer une méthode gloutonne, est d'ajouter les objets de valeurs élevées en premier, jusqu'à saturation du sac.
Cette méthode est parfois efficace, mais parfois pas, on verra ses limites dans la prochaine partie.
Prenons un exemple, afin d'illustrer cela.

Supposons qu'on dispose d'un sac de capacité W=26 et du set d'objets que voici

Objets	A	B	C	D	E	F	G	H	I	J	K	L	M	N
Valeurs	4	3	8	5	10	7	1	7	3	3	6	12	2	4
Poids	2	2	5	2	7	4	1	4	2	1	4	10	2	1

Set d'objets à notre disposition

Suivons le principe de la méthode et prenons les objets de meilleure valeur.
Ça nous donne le sous-set d'objets suivant :

L(12,10); E(10,7); C(8,5); F(7,4)

Notre sac est tout juste saturé et la somme des valeurs des objets qu'il contient est de 37.

> *Cette solution est-elle optimale ?*

Rien, *a priori*, ne garantit l'optimalité de cette solution. Nous verrons plus tard ce qu'il en est.

Implémentation

On va reprendre l'idée, le principe des algorithmes gloutons et déterminer des solutions locales au problème.

À partir d'une liste, dont les éléments sont des triplets [objet,valeur,poids], triée selon l'ordre décroissant des valeurs, on va remplir le sac à dos jusqu'à saturation.
On va ainsi parcourir tous les éléments à notre disposition via la liste et on ajoutera ceux dont le poids, cumulé aux poids des objets déjà dans le sac, est inférieur à la capacité totale du sac.
Lorsque la capacité totale est dépassée, ça signifie soit que le sac est plein, soit que l'objet qu'on veut insérer est trop lourd.

Il vient donc l'implémentation suivante :

```
def knapSack(totalWeight, objects):
        currentWeight = 0
        subSet = []
        for counter in range(len(objects):
                if currentWeight + objects[counter][-1] <= totalWeight:
                        currentWeight += objects[counter][-1]
                        subSet.append(objects[counter])
        return subSet

def retrivevingData():
        objects = input("Veuillez entrer le set d'objets sous la forme \"liste de liste\"
(e.g. [[objet1, valeur1, poids1],[objet2,valeur2,poids2]]) par ordre décroissant de valeur
:")
        totalWeight = input("Veuillez entrer la capacité maximale de votre sac à dos :")
        return knapSack(totalWeight, objects):

print "La proposition de la méthode gloutonne est :", retrievingData()
```

Notons que la terminaison de la boucle est garantie, puisqu'on itère sur une séquence finie.

Faiblesses des algorithmes gloutons

Les algorithmes gloutons présentent l'avantage d'une conception relativement aisée à mettre en oeuvre. Cependant, le prix à payer est qu'ils ne fourniront pas toujours la solution optimale au problème donné.

Revenons sur nos pas et réexaminons les exemples.

Le problème du rendu de pièces

L'algorithme glouton est d'une bonne efficacité avec le système monétaire européen, il fournit la solution à laquelle on s'attend et qui est optimale.

Maintenant, imaginons un système monétaire dans lequel il y aurait seulement des pièces de 1, 3 et 4 unités. Supposons qu'on doive me rentre 6 unités monétaires.

Tout le monde dira bien entendu qu'il faut rendre 2 pièces de 3 unités pour minimiser le nombre de pièces rendues. Cependant, la méthode gloutonne rendra 1 pièce de 4 unités et 2 pièces de 1 unité.
En effet, elle va d'abord choisir la pièce de plus haute valeur en dessous du rendu, soit celle de 4 unités. Il restera ensuite 2 unités à rendre, la seule possibilité étant de fournir 2 pièces d'une unité.

On aura donc 3 pièces au lieu de 2, la solution n'est pas optimale.

The 0-1 knapsack problem

Le même problème se pose avec le sac à dos.
Les poids que j'ai fourni dans notre exemple sont relativement équilibrés, de sorte que la solution fournie est probablement optimale.

Mais qu'en serait-il si les poids des objets étaient très déséquilibrés ?
Prenons un exemple, on dispose d'un sac à dos d'une capacité de 40 et du set d'objets suivant

Objets	A	B	C	D	E	F
Valeurs	30	12	12	12	12	4
Poids	39	10	10	10	10	1

Set d'objets à notre disposition
L'algorithme choisira l'objet A et l'objet F, ce qui fera une somme des valeurs de 34.

Pourtant, on remarque directement qu'en choisissant les 4 objets B,C,D,E; on aurait pu atteindre une somme des valeurs de 48, pour le même poids.
L'algorithme n'a pas non plus produit ici une solution optimale.

Comment faire pour avoir une solution optimale ?

Il n'y a pas de réponse miracle à cette question, tout dépend du problème.

Dans certains cas, les algorithmes gloutons produisent d'excellents résultats et sont appropriés au problème, dans d'autres cas, non.
Généralement, si les poids des objets sont très déséquilibrés, les algorithmes gloutons produiront une solution non optimale car de tels algorithmes ont une mauvaise vision globale du problème.

Les exemples qui ont été traités ici peuvent très bien être résolus à l'aide de la programmation dynamique (http://www.siteduzero.com/tutoriel-3-95368-introduction-a-la-programmation-dynamique.html) ou à l'aide du paradigme de programmation divide-and-conquer (http://www.siteduzero.com/tutoriel-3-58341-une-classe-d-algorithme-non-naifs-diviser-pour-regner.html).
On aurait pu également utiliser une solution de brute-force, mais ce genre de solutions est à éviter en raison de leur très mauvaise complexité algorithmique (http://www.siteduzero.com/tutoriel-3-51767-la-notion-de-complexite.html).

Il faut réfléchir à la solution à adopter en fonction du problème.

Voilà, vous en savez désormais plus sur les algorithmes gloutons.

Vous devriez être capables de les implémenter dans votre langage de programmation favori.

Pour toute remarque, vous pouvez m'envoyer un MP (http://www.siteduzero.com/mp-273-158415.html) !