Leçon d'informatique : algorithme glouton

S. Benlhajlahsen - PCSI$_1$

Sommaire

Extrait du programme :

Thèmes	Exemples d'activité, au choix du professeur et non exigibles des étudiants. Commentaires
Algorithmes gloutons.	Rendu de monnaie. Allocation de salles pour des cours. Sélection d'activité. *On peut montrer par des exemples qu'un algorithme glouton ne fournit pas toujours une solution exacte ou optimale.*

I Un exemple d'introduction

Durée d'un parcours sur un réseau ferroviaire : Imaginons un voyageur qui souhaite aller d'une gare de départ à une gare d'arrivée (voir 1). Celui-ci souhaite **optimiser** la durée de son parcours. Au moment de décider son chemin, le voyageur ne connaît que la durée qui le sépare des gares vosines. Pour différentes raisons, il ne connaît pas le temps entre les stations intermédiaires et la gare d'arrivée. Faisant le **meilleur choix à cet instant**, il décide de passer par la station 1 alors que le **meilleur choix global** aurait été de passer par la station 3.

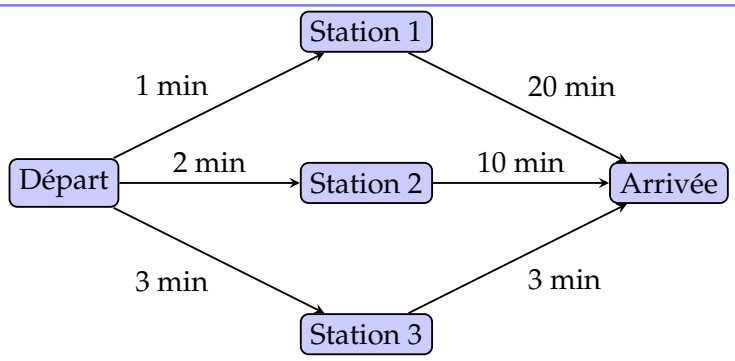

FIGURE 1

Remarque : Une solution possible eût été de « tester » chaque trajet puis de décider la bonne solution. On comprend que cette méthode appelée **force brute** prend énormément de temps même si elle donne la solution optimale.

II Définitions

À retenir : Un algorithme glouton [a] est un algorithme qui suit le principe de faire, à chaque étape, un choix optimal, dans l'espoir d'obtenir un résultat globalement optimal.
De plus, une fois le choix fait, on ne peut pas revenir en arrière.

a. *greedy algorithm* en anglais

Avantage d'une heuristique gloutonne : On parle d'heuristique car cette méthode de calcul fournit rapidement une solution réalisable, pas nécessairement optimale ou exacte.

Maximum local ou global :

Sur la figure 2, on veillera à bien distinguer un maximum local m du maximum global M. Imaginons un randonneur placé initialement en O qui souhaite minimiser ses efforts en prenant la route de plus faible pente.

En prenant la route de gauche, il fait le meilleur choix initial mais le plus mauvais choix au final : son effort sera un maximal global.

En prenant la route de droite, il fait le plus mauvais choix initial mais le meilleur choix au final : son effort sera un minimum global.

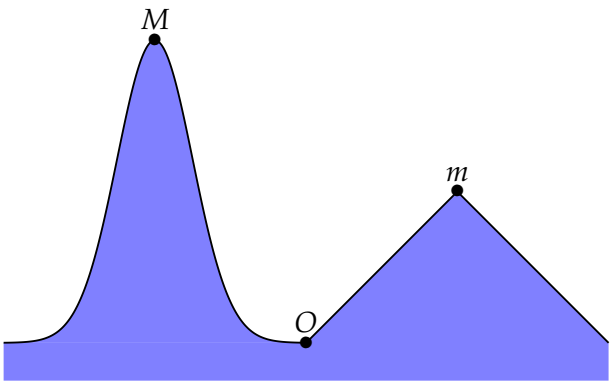

FIGURE 2

III Quelques implémentations

III.A Rendu de monnaie

Après un achat, un commerçant souhaite rendre la monnaie en espèce (pièces ou billets). Si on note `monnaie` un entier représentant la somme à rendre et `pieces` la liste des pièces et billets dans le pays, un algorithme glouton consistera à commencer par rendre le billet de plus forte valeur. Par exemple, si on souhaite rendre 472 € et qu'on dispose des pièces `[1,2, 5, 10, 20, 50, 100, 200, 500]`, le commerçant essaiera d'abord de rendre 2 billets de 200 € puis un billet de 50, puis un billet de 20 € et, finalement, une pièce de 2 €.

Version itérative : On écrit une fonction `rendu(pieces, monnaie)`. Dans la version itérative (voir figure 3), comme on ne connaît pas, par avance, le nombre de billets à rendre donc on privilégie une boucle `while`. La fonction renvoie une liste de même taille que `pieces` qui donne le nombre de pièces/billets rendus par type.
Ainsi, `rendu([1, 2, 5, 10, 50], 47)` renvoie `[0, 1, 1, 4, 0, 0]` car le commerçant donnera 4 billets de 10 €, une pièce de 5 € et une pièce de 2 €.

```
1  def rendu(pieces, monnaie):
2      n = len(pieces)
3      res = [0 for i in range(n)]
4      k = n-1
5      while monnaie>0 and k>=0:
6          if pieces[k] > monnaie:
7              k = k -1
8          else:
9              monnaie = monnaie - pieces[k]
10             res[k] = res[k] + 1
11     return res
12 pieces1 = [1,2,5,10,50,100]
13 print(rendu(pieces1, 47))                    # renvoie [0, 1, 1, 4, 0, 0]
```

FIGURE 3 – Algorithme de rendu de monnaie en version itérative.

Question 1 : Justifier la terminaison de cet algorithme.

Solution optimale ? Cette méthode donne une solution rapidement mais donne-t-elle un nombre minimal de pièces/billets ? Cela dépend de la liste `pieces`. En effet, `rendu([1, 3, 4], 6)` renverra `[2, 0, 1]` alors qu'une solution optimale eût été `[0,2, 0]`.
Par contre, avec les jeux de pièces **canoniques**[a], la méthode gloutonne sera toujours optimale.

a. jeu de pièces qui s'est standardisé dans le monde (euro, dollar, yen).

Version récursive : Pour la version récursive, il faut établir une relation de récurrence entre deux remise de pièce. Si on rend la pièce `pieces[k]`, `monnaie` est remplacé par `monnaie-pieces[k]` et on compte la pièce rendu. On propose une solution en figure 4 qui fait intervenir une fonction auxiliaire `ajouter` pour modifier la liste des pièces rendues.

```
1  def ajouter(k, L):# ajouter 1 en position k
2      L2 = [0 for i in range(len(L))]
3      for i in range(len(L)):
4          if i!=k:
5              L2[i] = L[i]
6          else:
7              L2[i] = L[i]+1
8      return L2
9  print(ajouter(1, [2,2]))                                    # renvoie [2, 3]
10
11 def rendu(pieces, monnaie):
12     if monnaie == 0:
13         return [0 for i in range(len(pieces))]
14     else:
15         k = len(pieces)-1
16         while pieces[k] > monnaie:
17             k = k-1
18         return ajouter(k ,rendu(pieces, monnaie-pieces[k]))
19 pieces1 = [1,2,5,10,50,100]
20 print(rendu(pieces1, 47))                          #  renvoie [0, 1, 1, 4, 0, 0]
```

FIGURE 4 – Algorithme de rendu de monnaie en version récursive.

III.B Problème du sac à dos

Présentation : On souhaite remplir un sac-à-dos à l'aide d'objets dont on connaît la masse et la « valeur ». Par exemple, dans l'exemple ci-dessous :

numéro de l'objet	0	1	2	3	4
valeur v_i de l'objet (en €)	15	60	10	7	10
masse m_i (en kg)	6	25	5	8	20
Rapport valeur/masse	2.5	2.4	2	0.875	0.5

On voit que l'objet numéroté 0 a une valeur de 15 € et une masse de 6 kilogrammes. On peut représenter cet ensemble d'objet par deux listes :

`V = [15, 60, 10, 7, 10]` et `M = [6, 25, 5, 8, 20]`

ou alors par une liste de liste :

`Objets = [[15,6], [60,25], [10,5], [7,8], [10,20]]`

Dans ce dernier exemple, `Objets[i][0]` et `Objets[i][1]` donnent respectivement la valeur et la masse du `i`-ème objet. On peut remplir le sac-à-dos en optimisant le rapport valeur/masse. On notera d'ailleurs que la liste `Objets` est triée selon un rapport valeur/masse décroissant. On commencera alors de remplir le sac-à-dos avec le premier objet puis le second… Bien évidemment, la masse totale du sac-à-dos est limitée à une valeur maximale M.

Remarque : La différence essentielle avec le rendu de monnaie est qu'on ne dispose que d'un seul objet de chaque type alors que, pour le rendu de monnaie, on pouvait, par exemple, rendre autant de billets de 100 € que nécessaire.

Remarque : Comme pour le rendu de monnaie, cet algorithme suppose que les listes soient triées. Ici, la liste `Objets` est triée selon le rapport valeur sur masse.

Version itérative : En figure 5, on donne une version itérative. On balaie les objets en nombre `len(Objets)` connu : on peut donc utiliser une boucle `for`.
La terminaison de l'algorithme est donc évidente et sa complexité est linéaire.

```
def sacados(Objets,M):
    v = 0 # valeur du sac
    m = 0 # masse du sac
    for i in range(len(Objets)):
        if m+Objets[i][1]<= M:
            v += Objets[i][0]
            m += Objets[i][1]
    return v
M1 = 30
Objets1 = [[15,6], [60,25], [10,5], [7,8], [10,20]]
print(sacados(Objets1, M1))
```

FIGURE 5 – Algorithme du sac-à-dos en version itérative.

Version récursive : En figure 6, on donne une version récursive. On peut note M_k la masse disponible avant de tenter de mettre le `k`-ième objets. On a évidemment $M_0 = M$ et M_k une suite décroissante positive.

```
def sacados(Objets,M):
    if M==0 or len(Objets)==0:
        return 0
    else:
        v = Objets[0][0]
        m = Objets[0][1]
        if m<=M:
            M = M-m               # M est ici la masse disponible dans le sac
            return v+sacados(Objets[1:], M)
        else:
            return sacados(Objets[1:], M)
M1 = 30
Objets1 = [[15,6], [60,25], [10,5], [7,8], [10,20]]
print(sacados(Objets1, M1))
```

FIGURE 6 – Algorithme du sac-à-dos en version récursive.

Solution optimale : Dans le scripts proposé, pour $M = 30$ kg, l'algorithme renvoie une valeur de 32 €. En effet, on met les objets 0, 1 et 3 pour un poids total de 19 kilogrammes. Cette solution n'est pas optimale car on aurait pu mettre exactement 30 kilogrammes pour une valeur de 70 € en mettant les objets 1 et 2.

Conclusion : On a vu que la méthode gloutonne fournit des solutions rapides pas nécessairement optimale. D'autres méthodes de programmation dynamique permette de trouver des solutions optimales.