ALGORITHMES GLOUTONS

Un problème d'optimisation classique consiste à maximiser ou minimiser une fonction tout en respectant une ou plusieurs *contraintes*. Prenons un exemple : on veut passer par 3 villes A, B, C (c'est la contrainte) en parcourant un minimum de distance (fonction à minimiser) pour économiser l'essence. On part du point O, cf le schéma ci-dessous (1 graduation = 1 km) :

Pour trouver le trajet optimal, on peut tester tous les trajets possibles et prendre le meilleur. On trouve que c'est $O \to B \to A \to C$, qui permet de passer par toutes les villes en parcourant seulement 9 km. Cependant, pour un problème plus compliqué, tester tous les cas possibles relève de l'impossible : le coût en calcul est trop grand.

L'objectif d'un *algorithme glouton* est d'obtenir une solution rapidement, mais cette solution n'est pas forcément optimale. À chaque étape de l'algorithme, on lui propose un ensemble de choix : un algorithme glouton va prendre le meilleur choix « à court terme ». Dans notre exemple, l'algorithme glouton va choisir à chaque étape d'aller à la ville la plus proche selon la position actuelle. Le trajet sera donc $O \to A \to B \to C$, pour un total de 11 km. Ainsi, *faire une suite de choix optimaux à court terme ne conduit pas forcément à la solution optimale sur le long terme.* Notons toutefois que si C était à gauche de B, l'algorithme glouton donnerait la solution optimale.

1 Problème du rendu de monnaie

Pour simplifier nous supposons que nous n'avons que des pièces. Un système de pièces est alors un n-uplet $P = (p_0, p_1, \ldots, p_{n-1})$, où p_i représente la valeur de la pièce d'indice i. Dans le système de la zone euro, par exemple, nous avons en centimes les pièces suivantes : $P = (1, 2, 5, 10, 20, 50, 100, 200, 500)$ où 100,200 et 500 représentent respectivement les pièces de 1,2 et 5 euros.

Le problème du rendu de monnaie consiste à rendre une somme d'argent r avec le minimum de pièces. En d'autres termes, il faut trouver une liste d'entiers positifs $[x_0, x_1, \ldots, x_{n-1}]$ qui vérifie $x_0p_0 + x_1p_1 + \ldots + x_{n-1}p_{n-1} = r$ en minimisant la somme $x_0 + x_1 + \ldots + x_{n-1}$, c'est-à-dire le nombre de pièces utilisées. La condition $x_0p_0 + x_1p_1 + \ldots + x_{n-1}p_{n-1} = r$ constitue la contrainte.

Supposons qu'on veuille rendre 8 centimes. On ne peut donc utiliser que les pièces de 1,2 et 5 centimes. On cherche donc les triplets $[x_0, x_1, x_2]$ possibles tels que $x_0 + 2x_1 + 5x_2 = 8$. Le programme suivant liste de manière exhaustive tous les triplets :

```
p=(1,2,5)                # p est un 3-uplet (ou triplet), même syntaxe qu'une liste
for x0 in range(9):
    for x1 in range(5):
        for x2 in range(2):
            if x0*p[0] + x1*p[1] + x2*p[2] == 8:
                print( [x0,x1,x2] )
```

Exercice 1. Exécuter le programme. Quel est le triplet qui minimise le nombre de pièces utilisées ?

Un algorithme glouton proposera la solution suivante. Si on veut rendre $r = 9$ euros :

- On regarde quelle est la pièce de plus haute valeur avant 9 : c'est 5. On la rend une fois. Il reste $9 - 5 = 4$ euros à rendre.
- On regarde quelle est la pièce de plus haute valeur avent 4 : c'est 2. On la rend une fois. Il reste $4 - 2 = 2$ euro à rendre.
- On regarde quelle est la pièce de plus haute valeur avant 2 : c'est 2. On la rend une fois. Il reste $2 - 2 = 0$ euro à rendre. Fini.

Dans cet exemple, on retourne ainsi 2 pièces de deux euros et 1 pièce de cinq euros. On attend donc que l'algorithme retourne `[0,2,1]`

Exercice 2. Compléter le script `monnaie(p,r)`ci-dessous qui retourne la solution de l'algorithme glouton pour un système de pièces `p` et une somme `r` à rendre. Tester si `monnaie( (1,2,5), 9)` retourne bien `[0,2,1]`.

```python
def monnaie(p,r):
    n = len(p)
    i = n-1
    solution = n*[0]      # tester n*[0] en console si besoin !
    while r>0:
        while p[i]>r:     # la pièce numéro i a une valeur trop grande
            ...
        ...
        ...
    return solution
```

Exercice 3. Tester différentes valeurs de r avec $p = (1,2,5,10,20,50)$. Est-ce que l'algorithme glouton vous semble optimal ?

Jusqu'en 1971, le système monétaire utilisé en Angleterre comportait une multitude de pièces : 1, 3, 4, 6 pence, 1 shilling (qui vaut 12 pence), etc.

Question 1. On veut rendre 8 pence avec ce système. La solution gloutonne vous semble-t-elle optimale ?

2 Stations d'essence

Voici un problème pour lequel l'algorithme glouton se débrouille bien. 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_{max} avec un plein. La route empruntée comporte n stations-services. 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 considéré comme une station, et est à une distance d_{n-1} de la station précédente.

L'automobiliste pourra effectuer son trajet si et seulement si $d_0, d_1, \ldots, d_{n-1} \leq d_{max}$. Nous supposons que cette condition est remplie. **L'objectif de l'automobiliste est de s'arrêter le minimum de fois dans une station-service.**

Un algorithme glouton va, lorsque le plein est fait, parcourir la plus grande distance possible sans faire le plein. La valeur d_{max} étant fixée, on va aller à la dernière station-service qui est atteignable sans faire le plein (s'il y a d'autres stations avant, on ne s'y arrêtera pas).

Par exemple, si $[d_0, d_1, \ldots, d_{n-1}] = [2,2,3,1,5,1]$ et $d_{max} = 5$, alors :

- On va jusqu'à la station numéro 1 car $d_0 + d_1 = 4 \leq d_{max}$ mais $d_0 + d_1 + d_2 = 7 > d_{max}$. On fait le plein à la station numéro 1.
- On va jusqu'à la station numéro 3 car $d_2 + d_3 = 4 \leq d_{max}$ mais $d_2 + d_3 + d_4 = 9 > d_{max}$. On fait le plein à la station numéro 3.
- On va jusqu'à la station numéro 4 car $d_4 = 5 \leq d_{max}$ mais $d_4 + d_5 = 6 > d_{max}$. On fait le plein à la station numéro 4.
- On va jusqu'à la station numéro 5 (la dernière c'est l'arrivée) car $d_5 = 1 \leq d_{max}$. Fin de l'algorithme.

On s'est arrêté aux stations 1, 3, 4, 5. Sur cet exemple, l'algorithme doit retourner :

```
>>> glouton( [2,2,3,1,5,1] , 5 )
[1,3,4,5]
```

```python
def glouton(distances,dmax):
    n = len(distances)
    stations = []      # numéros des stations où on s'est arrêté
    d = 0              # distance parcourue depuis la dernière station
    i = 0              # n° de la dernière station où on est passé, en s'arrêtant ou non
    while i!=n:
        while i<n and ...
            ...
            ...
        stations.append(i-1)  # le plein est fait
        d = 0
    return stations
```

Exercice 4. Compléter le programme ci-dessus et le tester avec l'exemple. L'algorithme glouton vous semble-t-il optimal ?

Idée de la preuve d'optimalité. On peut montrer qu'effectivement, l'algorithme glouton fournit toujours une solution optimale. L'heuristique de la preuve est la suivante. On considère :

- Une solution $s := [s_0, s_1, \ldots, s_{m-1}, s_m]$ obtenue avec l'algorithme glouton.
- Une solution optimale quelconque $S := [S_0, S_1, \ldots, S_p]$.

Comme S est optimale, on a nécessairement $p \leq m$. Prouvons que $p = m$, càd que la stratégie gloutonne fait aussi bien qu'une stratégie optimale donnée. Si les listes s et S sont égales, il n'y a rien à prouver. Supposons donc que $s \neq S$.

Soit k le plus petit entier tel que $s_k \neq S_k$. La solution S peut donc s'écrire $S = [s_0, s_1, \ldots s_{k-1}, S_k, S_{k+1}, \ldots, S_p]$. On montre alors que $S' := [s_0, s_1, \ldots s_{k-1}, s_k, S_{k+1}, \ldots, S_p]$ est elle aussi une solution, optimale puisqu'il y a $p+1$ arrêts comme pour S.

En itérant ce processus, on finit par arriver à la conclusion que $[s_0, s_1, \ldots, s_p]$ est une solution optimale. Alors, si $p < m$, on aurait une contradiction car on pourrait retirer les arrêts $p+1, \ldots, m$ de la liste s, et avoir encore une solution, ce qui est impossible par construction de l'algorithme glouton. Donc $p = m$.

3 Approfondissement : 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_{max}. 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 :

A) on choisit un objet qui a la valeur maximale,

B) on choisit un objet qui a le poids minimal,

C) on choisit un objet qui a le ratio valeur / poids maximal.

L'algorithme glouton consiste, à chaque étape, à choisir parmi ces objets celui qui représente le meilleur choix (selon le critère A), B) ou C)). Ce meilleur choix aura un poids qu'on note $P1$. Ensuite, nous recommençons parmi les objets de poids $p_i \leq P_{max} - P1$. Et ainsi de suite. Prenons un exemple : le sac à dos peut contenir 15Kgs. Les poids des objets sont en Kg, les valeurs en euro.

Objet	Valeur	Poids	Ratio Valeur / Poids
Vélo	126	14	9
Vase	80	8	10
Livre	32	2	16
Nourriture	20	5	4
Vêtements	18	6	3
Bouteille	5	1	5

Un objet est représenté par une liste comme `['Velo',126,14]`. On définit d'abord trois fonctions qui prennent en paramètre la liste de l'objet et renvoient respectivement la valeur, l'inverse du poids, et le ratio valeur / poids. Ainsi l'objet qui est le meilleur choix (selon le critère A), B) ou C)) est celui qui maximise la fonction correspondante.

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

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

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

On définit ensuite la fonction `glouton` qui prend en arguments une liste d'objets, le poids maximal P_{max} du sac à dos, et l'une des 3 fonctions ci-dessus, selon le choix A)–C).

La première chose à faire est de trier la liste d'objets selon le type de choix utilisé, par ordre décroissant. Cela se fait avec la fonction `sorted` : l'option `reverse = True` permet de trier par ordre décroissant (par défaut c'est croissant).

```
def glouton(liste,pmax,choix):
    copie = sorted( liste, key = choix, reverse = True )
    sac = []
    VAL = 0   # valeur totale dans le sac
    PDS = 0   # poids total dans le sac
    i = 0
    while i < len(liste) and PDS <= pmax:
        nom,val,pds = copie[i]
        if ...
            sac.append(nom)
            ...
            ...
        i = i+1
    return sac,VAL
```

Enfin, il ne reste plus qu'à définir la liste d'objets puis à exécuter la fonction `glouton` en précisant le choix utilisé.

```
objets = [ ['Velo',126,14], ['Vase',80,8], ['Livre',32,2], ['Nourriture',20,5],
    ['Vetements',18,6], ['Bouteille',5,1] ]

print( glouton(objets,15,valeur) )
print( glouton(objets,15,invpoids) )
print( glouton(objets,15,ratio) )
```

Exercice 5. Compléter, compiler, admirer.

Question 2. On voit que choisir son objet selon le critère « `valeur` » est le plus intéressant puisque la valeur totale est 131. Mais est-ce une solution optimale ?