Algorithmes gloutons

Cyril Charignon

Table des matières

Première partie

Cours

1 Principe

Les algorithmes gloutons sont une catégorie d'algorithme utilisables pour la recherche d'un maximum (ou d'un minimum, ce qui revient au même en changeant la relation d'ordre). Exemple : chercher un enchaînement de tâches qui minimise le temps total, un plus court chemin, un choix d'objets à fabriquer et vendre pour obtenir le plus gros bénéfice possible, trouver le moyen de mettre le maximum de cours dans un même lycée...

Notons E l'ensemble de toutes les possibilités et $f : E \to \mathbb{R}$ la fonction dont nous voulons trouver le maximum.

Pour un tel problème, on peut envisager en général au moins trois méthodes générales :

- Force brute : calculer $f(x)$ pour tout $x \in \mathbb{R}$ et prendre le maximum. Praticable si le nombre de possibilités reste faible.
- Algorithme glouton : faire des choix successifs et à chaque étape prendre le choix qui semble optimiser f, sans réfléchir aux choix suivants.
- Programmation dynamique : ramener le calcul du maximum de f sur E à des calculs de maximum sur des sous-ensembles de E, et trouver une relation de récurrence. Sera étudié en deuxième année.

Exemples d'algorithmes gloutons dans la vraie vie :

- Au échecs : jouer à chaque tour le coup qui permet de prendre une pièce adverse la plus puissante possible.
- En rando : avancer toujours tout droit vers l'objectif.
- Vous avez plusieurs courses à faire en différents endroits de la ville. Vous allez toujours en priorité à l'endroit le plus proche restant à visiter.

Avantages et inconvénients :

- Facile à calculer
- Résultat moins pire qu'un choix aléatoire
- Résultat en général pas optimal

Ainsi en général un algorithme glouton est un compromis : une solution pas optimale, mais pas non plus trop mauvaise, obtenue assez facilement. Il existe cependant des situations dans lesquelles un algorithme glouton fournit la solution optimale.

2 Exemples

2.1 Rendu de monnaie

Une situation de la vie de tous les jours où on utilise généralement une méthode gloutonne et l'utilisation de pièces de monnaie pour payer une somme.

Notons s l'ensemble des pièces disponibles (le « système monétaire ») et n la somme à payer. La méthode gloutonne consiste à commencer par utiliser la plus grande pièce $x \in s$ de valeur inférieure à n, puis recommencer pour payer la somme restante, à savoir $n - x$.

Avec le système monétaire de l'euro, on peut démontrer que l'algorithme glouton donne toujours le résultat optimal. En revanche avec le système monataire anglais d'avant 1960, ce n'était pas le cas.

2.2 Emploi du temps

Dans un festival vous avez identifié la liste des concerts que vous aimeriez voir. Notons N le nombre de ces concerts et $\big((d_i, f_i)\big)_{i\in[\![0,n[\![}$ les instants de début et fin de ceux-ci. Vous ne souhaitez voir que des concerts entiers : si deux se chevauchent il faut faire un choix. Votre but est de voir un maximum de concerts.

Sur cet exemple, si on souhaite mettre en œuvre un algorithme glouton, plusieurs choix sont possibles pour la quantité à optimiser à chaque étape.

Imaginons différents critères à optimiser à chaque étape :

- Choisir en priorité le plus petit intervalle. Échoue sur cet exemple : `[(0,6), (5,7), (6,10)]`
- Prendre celui qui commence le plus tôt en premier. Échoue sur `[(0,2), (1,8)]`
- Choisir le concert ayant le moins d'interférences avec les autres en priorité. Échoue sur `[(0,3), (3,7),`
  `↪ (7,11), (11,15), (2,5),(2,5),(2,5), (6,8), (10,12),(10,12),(10,12)]`

Il existe une méthode qui donne toujours un résultat optimal : c'est de choisir en priorité le concert qui finit le plus tôt.

Étape de la preuve : soit $n \in \mathbb{N}$ et $(C_0, ..., C_{n-1})$ les concerts renvoyés par l'algorithme. Soit $m \in \mathbb{N}$ et $(D_0, ..., D_{m-1})$ une solution optimale, triée par ordre croissant de fin.

Pour tout concert X on notera $f(X)$ son heure de fin et $d(X)$ son heure de début.

1. On montre par récurrence que pour tout $k \in [\![0, \min n, m[\![$, $f(C_k) \leq f(D_k)$.
   - *Initialisation :* C_0 est par construction le concert qui finit le plus tôt. Donc $f(C_0) \leq f(D_0)$.
   - *Hérédité :* Soit $k \in [\![0, m[\![$, supposons $\forall i \in [\![0, k[\![$, $P(i)$, et prouvons $P(k)$.
     On a $f(D_k) \geqslant d(D_k) \geqslant f(D_{k-1}) \geqslant f(C_{k-1})$ (car D_k n'est pas en conflit avec C_k, puis par $P(k)$). Donc D_k n'est pas en conflit avec $C_0, ..., C_{k-1}$. Or notre algo a choisi pour C_k celui parmi les concerts compatibles avec $C_0, ..., Ck - 1$ qui a l'heure de fin la plus précoce. Donc $f(C_k) \leq f(D_k)$.
2. Supposons $n < m$:

   > Alors D_n est un intervalle compatible avec $C_0, ..., C_{n-1}$. Ceci est absurde car l'algorithme s'arrête lorsqu'il ne reste aucun intervalle compatible avec ceux déjà pris.

   Ainsi $n \geqslant m$.
3. Conclusion : Comme D était une solution du problème, on avait aussi $m \geqslant n$. Donc $n = m$ et C est bien une liste de concerts compatibles entre eux de cardinal maximal.

Remarque : Si on introduit une notion d'importance entre les concerts (à durée égale on préfère certains concerts à d'autres), l'algorithme glouton n'est plus optimal et il faut recourir à une méthode de « programmation dynamique », attendre la seconde année !

Deuxième partie

Exercices

Exercice

Exercice 1. ** Gendarmes et voleurs

Un tableau est rempli de `"G"` et de `"V"`, le premier désignant un gendarme et le second un voleur. On fixe également un entier k. Chaque gendarme peut attraper au plus un voleur et celui-ci doit être situé à au plus k cases de sa position initiale.

Le but est de calculer le nombre maximal de voleurs pouvant être attrapés. On va utiliser un algorithme glouton.

1. Montrer que la stratégie « chaque gendarme de gauche à droite attrape le voleur le plus proche possible » échoue sur certains exemples.
2. Montrer que la stratégie « chaque gendarme de gauche à droite attrape le voleur le plus loin possible » échoue aussi sur certains exemples.
3. La méthode qui fonctionne (mais on ne demande pas de le prouver) est celle-ci : choisir à chaque étape le couple (i, j) admissible (en case i il y a un voleur, en case j un gendarme, et $|j - i| \leq k$) pour lequel $\big(\min(i, j), i, j\big)$ est minimal.
   (a) Programmer cette méthode.
   (b) La tester un particulier sur les exemples trouvés aux questions précédentes.

Problème

1 Introduction

Un voleur pénètre dans la maison d'un riche trader. Les objets les plus intéressants sont une collier de perles, un candelabre en argent, le livre « algorithmes » de Cormen, Leiserson, Rivest, Stein dédicacé par les auteurs, et une fourrure de glomorphe à rayures. Comme il doit rester léger pour escalader la façade de l'immeuble, il n'a pris qu'un petit sac à dos pouvant contenir au maximum 8kg. Le tableau suivant donne le poids et la valeur (en centaines d'euros) des différents objets :

objet	poids (kg)	valeur (100€)
Collier	1	15
Candelabre	5	10
Cormen	3	9
Glomorphe	4	5

Le voleur s'interroge sur les objets qu'il doit mettre dans son sac à dos. On envisage trois stratégies :

1. *Le voleur est glouton :* L'algorithme glouton consista à prendre en priorité les objets de plus grande valeur. Quels objets prendra-t-il s'il suit cette méthode ? Le choix est-il optimal ?
2. *Le voleur a le temps :* Supposons que le voleur décide de caculer toutes les combinaisons d'objet possibles. Il lui faut 5s pour analyser chaque possibilité (voir si elle est possible et calculer combien elle lui rapporte). Combien de temps lui faudra-t-il ?
3. *Le voleur est informaticien :* Il utilise un algorithme de programmation dynamique pour calculer la solution optimale. Ce sera vu en fin de problème.

2 Notations

On considére un sac à dos dont la capacité est notée C. On considère $n \in \mathbb{N}$ et n objets, dont on note $\rho_0, ..., \rho_{n-1}$ les poids et $v_0, ..., v_{n-1}$ les valeurs. Les poids et les valeurs sont des entiers strictement positifs [1]. Dans les programmes, ces valeurs seront rentrées dans deux tableaux `rho` et `v`.

Ainsi le but est de choisir en sous-ensemble X de $[\![0, n[\![$ tel que $\sum_{k \in X} \rho_k \leqslant C$ et pour lequel la valeur totale $\sum_{k \in X} v_k$ soit maximale.

1. L'algo serait plus compliqué si on autorisait des poids flottants.

3 Algorithme glouton

L'algorithme glouton est en réalité un peu plus fin que de prendre en priorité les objets ayant la plus grande valeur. Il s'agit de prendre en priorité les objets ayant le meilleur rapport valeur/poids. Pour tout $i \in [\![0, n[\![$, on notera $\alpha_i = \frac{v_i}{\rho_i}$ ce rapport.

1. Écrire une fonction `rapport_valeur_poids` prenant `rho` et `v` et renvoyant le tableau $[\alpha_0, \alpha_1, ..., \alpha_{n-1}]$.

On va utiliser un tableau `pris` rempli de booléens pour enregister les objets pris. Pour tout $i \in [\![0, n[\![$, `pris[i]` sera vrai si et seulement si l'objet i a été mis dans le sac à dos.

2. Écrire une fonction `meilleurObjetRestant` qui prend en entré le tableau `alpha` des α_i ainsi que le tableau `pris` et qui renvoie le numéro l'objet de meilleur rapport valeur/poids parmis les objets pas encore placés dans le sac à dos.

   Si tous les objets sont pris, on renverra -1.
3. En déduire la fonction finale. Elle renverra la liste des objets pris, ainsi que la valeur totale de ces objets.
4. M. X teste son programme avec les données de 1, le programme indique de prendre le collier et le Cormen. Ce qui est quelque peu idiot puisqu'il y a encore la place pour la fourrure de glomorphe à rayures. Pourquoi ce résultat ? Comment éviter ce problème ?

```
αρρραααραααααα
def glouton(v, ρ, c):
    n = len(v)α
     = rapport_valeur_poids (v, ρ)
    pris = [ False for i in range (0,n)]
    res = []
    c_restant = c
    fini = False

    while not fini :
        i = meilleur_objet_restant(pris, α)
        if i != -1 and c_restant >= ρ[i]:
            res.append(i)
            pris[i] = True
            c_restant -= ρ[i]
        else:# Plus d'objet, ou l'objet restant est trop lourd.
            fini = True

    return res
```

5. Trouver un exemple ou l'algorithme glouton ne fournit pas la solution optimale.

   Indication : Le fait d'utiliser le rapport valeur/poids au lieu de simplement la valeur fait que l'exemple de 1 n'est plus un exemple où l'algorithme glouton n'est pas optimal.

   On peut par exemple prendre un cas à dos de capacité 8, et trois objets de poids 5, 4, et 4. Faire en sorte que la solution optimale soit de prenddre les deux objets de poids 4, mais que l'algorithme glouton choisisse l'objet de poids 5.
6. Compter le nombre de comparaisons entre éléments du tableau `alpha` lors d'exécution de votre programme.
7. (***) La fonction **`sorted`** de Python prend un tableau et renvoie un tableau contenant les mêmes éléments mais dans l'ordre croissant. Elle nécessite environ $n \log_2(n)$ comparaisons pour un tableau de longueur n.

   Utiliser ceci pour améliorer votre fonction. Vérifier en comptant le nouveau nombre de comparaison que votre fonction est effectivement améliorée.

4 Programmation dynamique

Ici nous noterons C_0 la capacité du sac.

Pour tout $C \in [\![0, C_0]\!]$, et tout $i \in [\![0, n]\!]$, on pose $V(C, i)$ la valeur maximale des objets qu'on peut mettre dans un sac de capacité C en choisissant uniquement des objets parmi les i premiers (soit dans $[\![0, i[\![$). Si $C \leqslant 0$, nous convenons que $V(C, i) = 0$.

1. Que vaut $V(C, i)$ lorsque $C = 0$ ou $i = 0$?

2. Démontrer que pour tout $C \in [\![1, C_0]\!]$ et $i \in [\![0, n[\![$,

$$V(C, i+1) = \max\left(V(C, i),\ V(C-\rho_i, i) + v_i\right)$$

À présent, soit $C_0 \in \mathbb{N}$. La stratégie est de créer une matrice `V` de format $(C_0+1, n+1)$ telle que pour tout $(c, i) \in [\![0, c]\!] \times [\![0, n]\!]$, `V[c][i]` contiendra $V(c, i)$.

On suppose connue une fonction `nouvelle_matrice` prenant deux entiers p et q ainsi qu'un élément x qui renvoie une matrice de format (p, q) remplie de x.

3. Écrire une fonction prenant C_0, v, *rho* et renvoyant la valeur maximale des objets qu'on peut mettre dans un sac de capacité C_0. L'algorithme utilisé consistera à créer puis remplir la matrice `V` grâce à la formule de la question précédente.

   Indication : Suivre exactement la formule ci-dessus. Prendre en particulier garde aux valeurs de c et de i pour lesquelles elle est valide.

4. Calculer la complexité de cette fonction. Comparer cette complexité à celle de la méthode naïve qui consiste à essayer toutes les combinaisons possibles d'objets.

5 Questions difficiles

1. La méthode précédente est dite « de bas en haut » car on calcule $V(c, i)$ en commençant par les petites valeurs de c et de i. Un défaut de cette méthode est qu'on remplit certaines cases de `V` qui ne seront pas utiles ensuite. Une méthode « de haut en bas » consiste à partir de la case qu'on souhaite remplir (`V[c0,n]`) et de ne remplir que les cases nécessaires pour remplir celle-ci. Elle nécessite une fonction auxiliare récursive, c'est-à-dire qui se rappelle elle-même.

   Compléter le code suivant pour implémenter cette méthode.

```
def sac_à_dos_mémo(c0, v, rho):
    n = len(v)
    V = nouvelle_matrice(C0+1, n+1, -1) # -1 signifiera « case pas encore remplie »

    # Initialisation : mettre des 0 dans la première ligne et la première colonne
    ...

    def aux(c, i):
        """ Renvoie V(c,i) et remplit la case correspondante si elle ne l'était pas
            ↪ encore. """
        if c<=0:
            return 0
        elif V[c][i] != -1:
             return V[c][i]
        else:
            ...


    # Résultat final
    return aux(c0, n)
```

2. **Reconstruction de la solution optimale :** Le programme précédent ne calcule que la valeur maximale des objets qu'on peut mettre dans le sa à dos, et non la liste de ces objets. On peut employer deux méthodes pour obtenir cette dernière. La première consiste modifier le programme précédent pour remplir simultanément à la matrice `V` une matrice `listesObjets` telle que pour tout $c, i \in [\![0, C_0]\!] \times [\![0, n]\!]$, `listesObjets[c][i]` contiendra la liste des objets, choisis dans $[\![0, i[\![$, pour remplir au mieux un sac de capacité c. La seconde consiste à retrouver cette liste après coup en lisant le tableau `V` obtenu. L'idée est que si $V(c, i+1) = V(c, i)$ cela signifie que l'objet i n'a pas été pris. Et réciproquement, si $V(c, i+1) = V(c-\rho_i, i) + V_i$ c'est que l'objet i a été pris.

   Programmer une fonction prenant en entrée la matrice `V` remplie et renvoyant la liste des objets à prendre.