Algorithmes gloutons

INFORMATIQUE COMMUNE - TP nº 13 - Olivier Reynet

À la fin de ce chapitre, je sais :

- ☞ expliquer le principe d'un algorithme glouton
- ☞ reconnaître les cas d'utilisation classiques des algorithmes gloutons
- ☞ coder un algorithme glouton en Python
- ☞ détecter des cas de non-optimalité des solutions

A Occupation d'un salle de spectacles

On dispose d'une salle de spectacles et de nombreuses demandes d'occupation ont été faites le même jour, pour des spectacles différents. On a recensé ces spectacles dans une liste de tuples `L` contenant pour chaque spectacle le couple d'entiers `(d,f)` où `d` désigne l'heure de début et `f` l'heure de fin du spectacle. Deux spectacles ne peuvent pas avoir lieu simultanément. Deux spectacles sont programmables à partir du moment où l'heure de début de l'un est supérieure ou égale à l'heure de fin de l'autre. On cherche à maximiser le nombre de spectacles dans la salle mais pas forcément le temps d'occupation de la salle.

L'idée gloutonne est de choisir [1] les spectacles qui se terminent les plus tôt afin d'en programmer un maximum. Tous les spectacles n'étant pas compatibles, ils ne seront donc pas tous programmés.

A1. Appliquer à la main un algorithme glouton à la liste de spectacles `[(0, 2), (1, 3), (2, 4), (1, 5), (3, 6), (4, 7), (5, 9), (6, 11), (9, 12)]`. **Cette liste a été triée par ordre croissant d'heure de fin et c'est nécessaire au bon fonctionnement de l'algorithme!** On prendra le premier élément de la liste comme premier spectacle planifié.

> **Solution :** On trouve : [(0, 2), (2, 4), (4, 7), (9, 12)].

A2. Écrire une fonction gloutonne pour planifier ces spectacles dont le prototype est `planify(L)`, où `L` est la liste des spectacles triée par ordre croissant d'heure de fin qui renvoie la liste des spectacles planifiés représentés par leur tuple.

A3. Tester cette fonction sur la liste `[(0, 2), (1, 3), (2, 4), (1, 5), (3, 6), (4, 7), (5, 9), (6, 11), (9, 12)]`.

Dans le cours, on montre que cette stratégie gloutonne adoptée est optimale, c'est-à-dire qu'elle renvoie bien le nombre maximal de spectacles que l'on peut organiser.

1. si possible. . .

Solution :

Code 1 – Planification de l'occupation d'une salle

```python
def planify(S):
    assert len(S) > 0
    planning = [S[0]]  # first spectacle
    h_end = S[0][1]
    for start, end in S:
        if h_end <= start:  # is it a solution ?
            planning.append((start, end))
            h_end = end
    return planning


#MAIN PROGRAM
spectacles = [(0, 2), (1, 3), (2, 4), (1, 5), (3, 6), (4, 7), (5, 9), (6, 11),
    (9, 12)]
print(planify(spectacles))
```

B Rendre la monnaie

Un commerçant doit à rendre la monnaie à un client. La somme à rendre est un somme entière m et le commerçant cherche à utiliser le moins de billets et de pièces possibles. On considère qu'il dispose d'autant de pièces et de billets qu'il le souhaite parmi le système monétaire euro.

B1. En utilisant un algorithme glouton, coder une fonction itérative qui renvoie la monnaie d'après le système monétaire euro dont le prototype est `give_change(m, V)` où `m` est de type `int` et `V` la liste des pièces et billets du système monétaire `V =[500,200,100,50,20,10,5,2,1]` **triée par ordre décroissant des valeurs.** Le résultat de cette fonction est une liste de tuples comportant le nombre et la valeur de la pièce ou du billets utilisés `(n,value)`. Par exemple, pour `m=83`, on obtient `[(1, 50), (1, 20), (1, 10), (1, 2), (1, 1)]`.

B2. Tester le code avec différentes valeurs. Le résultat obtenu est-il toujours optimal, c'est à dire présente-t-il toujours un minimum de pièces et de billets ?

B3. Coder une fonction récursive `rec_give_change(m, V)` équivalente à la fonction précédente.

B4. On peut montrer qu'avec notre système monétaire usuel, l'algorithme glouton renvoie toujours une solution optimale. Si l'on considère le système `[30,24,12,6,3,1]` et que l'on veut rendre 49, que renvoie l'algorithme glouton ? Est-il optimal ?

Solution :

Code 2 – Rendre la monnaie

```python
def give_change(m, V):
    to_give = m
    solution = []
    for v in V:  # greatest value first
        n = to_give // v  # how many times ?
```

```
        if n > 0:  # if 0, no solution with c
            solution.append((n, v))  # memorize
            to_give = to_give - n * v  # continue...
    if to_give == 0:  # success
        return solution
    else:
        return None  # no solution


def rec_give_change(m, V):
    if len(V) == 0 or m ==0:  # Stop condition
            return []
    else:
        v = V[0]    # choose greatest value
        n = m // v  # how many times ?
        if n > 0:   # if 0, no solution with v
            return [(n,v)] + rec_give_change(m - n * v, V[1:])  # continue...
        else :
            return rec_give_change(m - n * v, V[1:])  # continue...



#MAIN PROGRAM
V = [500, 200, 100, 50, 20, 10, 5, 2, 1]
for change in [23, 49, 83, 117, 199, 201, 322, 497]:
    print(change, give_change(change, V))
    print(change, rec_give_change(change, V))
    assert give_change(change, V) is not None
    assert rec_give_change(change, V) is not None
    assert give_change(change, V) == rec_give_change(change, V)


V = [30, 24, 12, 6, 3, 1]
change = 49
print(change, give_change(change, V))  # pas optimal -->  2 x 24 +1
print(change, rec_give_change(change, V))  # pas optimal --> 2 x 24 +1
```

C Remplir son sac à dos

On cherche à remplir un sac à dos. Chaque objet que l'on peut insérer dans le sac est **insécable** [2] et possède une valeur et un poids connu. On cherche à maximiser la valeur totale emportée dans la sac à dos tout en limitant [3] le poids à `max_weight`.

On dispose de plusieurs objets de valeur et de poids modélisé dans une liste de tuples
`objects=[(100, 40), (700, 15), (500, 2), (400, 9), (300, 18), (200, 2)]`
non ordonnée. `objects[i][0]` désigne la valeur de l'objet i et `objects[i][0]` son poids.

C1. Coder une fonction gloutonne et itérative pour résoudre ce problème. Son prototype est `knapsack (objects, max_weight)` où `max_weight` est de type `int`. Elle renvoie la liste des objets introduits

2. Soit on le met dans le sac, soit on ne le met pas. Mais on ne peut pas en mettre qu'une partie.
3. On accepte un poids total inférieur à `max_weight`.

dans le sac représentés par le tuple associé à l'objet `(v,p)`, la valeur totale cumulée qu'ils représentent ainsi que le poids total du sac ainsi obtenu. Par exemple, pour la liste d'objets `[(100, 40), (700, 15), (500, 2), (400, 9), (300, 18), (200, 2)]` et un poids maximal admissible de 44, la fonction renvoie `[(700, 15), (500, 2), (400, 9), (300, 18)], 1900, 44`.

C2. Au lieu de prendre l'objet de plus grande valeur, on prend celui de plus grand rapport valeur/-poids. Coder une fonction gloutonne et itérative qui implémente cette stratégie. Son prototype est `ratio_knapsack(objects, max_weight)`.

C3. Comparer les deux stratégies précédentes pour des poids maximums allant de 11 à 17 kg. Fournissent-elles toujours un résultat identique ? Lorsque le résultat n'est pas identique, une des stratégies fournit-elle la solution optimale ? Est-ce toujours la même qui fournit cette solution optimale ? Conclure.

Solution : Aucune de ces deux stratégies n'est optimale. Par exemple :

- pour un poids maximal de 11 kg, la première donne une valeur de 900 et la seconde 700. La valeur optimale est 900.
- pour un poids maximal de 15 kg, la première donne une valeur de 700 et la seconde 1100. La valeur optimale est 1100.

Ces algorithmes donnent donc parfois la solution optimale, mais pas toujours.

Code 3 – Sac à dos

```python
def greedy_kp(objects, max_weight):
    total_weight = 0
    total_value = 0
    pack = []
    objects = sorted(objects, reverse=False)  # to choose max val
    while len(objects) > 0 and total_weight <= max_weight:
        v,p = objects.pop()  # choose an object, the last object and the greatest value !
        if total_weight + p <= max_weight:  # is it a solution ?
            pack.append((v,p))  # memorize
            total_weight += p
            total_value += v  # and keep on
    return pack, total_value, total_weight


def ratios_knapsack(objects, max_weight):
    total_weight = 0
    total_value = 0
    pack = []
    ratios = sorted([(v / w, v, w) for v, w in objects], reverse=False)  # to choose max val
    # print(f"Ratios {ratios}")
    while len(ratios) > 0 and total_weight <= max_weight:
        r,v,p = ratios.pop()  # choose an object, the last object and the greatest value !
        if total_weight + p <= max_weight:  # is it a solution ?
            pack.append((v, p))  # memorize
            total_weight += p
            total_value += v  # and keep on
    return pack, total_value, total_weight

```

```python

#MAIN PROGRAM
o = [(100, 40), (700, 15), (500, 2), (400, 9), (300, 18), (200, 2)]
print(o)

for mw in range(11, 17, 1):
    gkp = greedy_kp(o, mw)
    rgkp = ratios_knapsack(o, mw)
    if gkp[1] != rgkp[1]:
        print("Max weight -->", mw)
        print("\tSame weight ? ", gkp[2] == rgkp[2], " -->  ", gkp[2], " vs", rgkp[2])
        print("\tSame value ? ", gkp[1] == rgkp[1], " --> ", gkp[1], " vs", rgkp[1])
    else:
        print("Max weight -->", mw)


o = [(10, 9), (7, 12), (3, 7), (2, 5), (1, 2)]
print(greedy_kp(o, 15))
print(ratios_knapsack(o, 15))


def kp_dp(v, w, W):
    n = len(w)
    s = []
    for i in range(n + 1):
        s.append([0 for _ in range(W + 1)])
        for p in range(W + 1):
            if i == 0:
                s[i][p] = 0  # no objects, no solution, value is zero
            elif w == 0:  # O kg, one solution : take 0 object of 0 value
                s[i][p] = 0
            elif w[i - 1] <= p:
                s[i][p] = max(v[i - 1] + s[i - 1][p - w[i - 1]], s[i - 1][p])
            else:
                s[i][p] = s[i - 1][p]
    return s[n][W]

print(kp_dp([10,7,3,2,1],[9,12,7,5,2], 15))

for v,w in o:
    print(v/w)
```

D Découper d'une barre de métal

On considère une barre de métal de longueur `total_length` de type `int`. La vente à la découpe procure des revenus différents selon la longueur de la découpe. On cherche à calculer le prix optimal que l'on peut obtenir de cette barre en la découpant à des abscisses entières. Il est possible de découper la barre plusieurs fois à la même longueur.

On dispose d'une liste de tuples `V` répertoriant les prix de vente des différentes longueurs : `V[i][0]` contient le prix de vente et `V[i][1]` la longueur associée.

D1. Écrire une fonction gloutonne pour découper de la barre en maximisant la valeur qui en résulte d'après le calcul rapport prix/longueur. Cette fonction a pour prototype :
`greedy_cut(V, total_length)`.

D2. Tester la fonction sur la liste `[(14, 3), (22, 5), (16, 4), (3, 1), (5, 2)]` pour une barre de longueur 5, la solution retournée dans ce cas semble-t-elle optimale ?

Solution :

Code 4 – Découper la barre

```python
def greedy_cut(V, total_length):
    ratios = sorted([(v / l, v, l) for v, l in V], reverse=False)  # from lower to higher prices
    print(ratios)
    remaining_length = total_length
    total_price = 0
    S = []  # solution
    while len(ratios) > 0 and remaining_length > 0:
        ratio, higher_price, length = ratios.pop()  # choose the best ratio, the last
        n = remaining_length // length  # how many times ?
        if n > 0 and length <= remaining_length:  # is it a solution ?
            S.append((higher_price, length, n))
            total_price += n * higher_price
            remaining_length -= n * length
    return S, remaining_length, total_price


#MAIN PROGRAM
V = [(14, 3), (22, 5), (16, 4), (3, 1), (5, 2)]
length = 5
gc = greedy_cut(V, length)
print(length, gc)
```

R Cet exercice présente un problème similaire à la variante du sac à dos étudiée plus haut à la question C2. C'est tout l'intérêt de la description algorithmique des problèmes : généraliser les résolutions. Seul le contexte et les mots avec lesquels on décrit le problème diffèrent. L'algorithme de résolution reste le même.

E Allouer des salles de cours (bonus)

Un proviseur adjoint cherche à allouer les salles de cours de son lycée en fonction des cours à programmer. Deux cours ne peuvent pas avoir lieu en même temps dans une même salle. On cherche le nombre minimal de salles à réserver pour que tous les cours aient lieu.

On modélise un cours par un tuple constitué du nom du cours et de la plage horaire du cours comme suit : `("Informatique", (11, 13))`. On dispose d'une liste de cours `lectures` à planifier dans des salles numérotées de 0 à N. L'algorithme peut créer autant de salles que nécessaire.

E1. Proposer un algorithme glouton de résolution de ce problème et l'appliquer à la liste
`lectures = [("Maths", (9, 10.5)), ("Info", (9, 12.5)), ("Info", (11, 13)), ("Maths", (11, 14)), ("Maths", (13, 14.5)), ("Maths", (8, 9.5)), ("Phys.", (10, 14.5)), ("Phys.", (16, 18.5)), ("Ang.", (13, 14)), ("Fr.", (10, 12))]`

E2. Écrire une fonction gloutonne de prototype `find_rooms(lectures)` implémentant cet algorithme et qui renvoie une liste dont les éléments sont des listes de tuples. L'indice de chaque liste dans la liste est le numéro de la salle de cours et les tuples contiennent les cours qui ont lieu dans cette salle. Par exemple : `[[('Maths', (8, 9.5)), ('Fr.', (10, 12))], [('Info', (9, 12.5))], [('Maths', (9, 10.5)), ('Maths', (11, 14))], [('Phys.', (10, 14.5))]]` signifie que dans la salle numéro 0 auront lieu un cours de mathématiques et un cours de français, dans la salle numéro 1 un cours d'informatique...

E3. Que pensez-vous du nombre de salles nécessaires ?

Solution :

Code 5 – Allocation de salles de cours

```python
def allocate_rooms(lectures):
    lectures = sorted(lectures, key=lambda tup: tup[1][0], reverse=True)
    # print(lectures)
    planning = []
    while len(lectures) > 0:  # there are lectures to plan
        title, (start, end) = lectures.pop()  # take the next lecture (from starting hour)
        # print("Dealing with --> ", title, (start, end))
        room = 0
        placed = False
        while room < len(planning) and not placed:  # Is there place in this room ?
            # print("\t\tstudying planning room", planning[room])
            if start >= planning[room][-1][1][1]:
                planning[room].append((title, (start, end)))
                placed = True
            else:
                room += 1  # search place in the next room
        if not placed:  # failing to place this lecture
            planning.append([])  # creating a new room
            planning[-1].append((title, (start, end)))
        # print("\tPlanning --> ", planning)
    return planning


#MAIN PROGRAM
L = [("Maths", (9, 10.5)), ("Info", (9, 12.5)), ("Info", (11, 13)), ("Maths", (11, 14)), ("Maths", (13, 14.5)),
     ("Maths", (8, 9.5)), ("Phys.", (10, 14.5)), ("Phys.", (16, 18.5)), ("Ang.", (13, 14)),
     ("Fr.", (10, 12))]

planning = allocate_rooms(L)
print("#",len(planning), "rooms are needed !")
print(planning)
```

```

# #5 rooms are needed !
# [[('Maths', (8, 9.5)), ('Fr.', (10, 12)), ('Ang.', (13, 14)), ('Phys.', (16,
    18.5))], [('Info', (9, 12.5)), ('Maths', (13, 14.5))], [('Maths', (9, 10.5))
    , ('Maths', (11, 14))], [('Phys.', (10, 14.5))], [('Info', (11, 13))]]
```