Algorithmes gloutons

INFORMATIQUE COMMUNE - TP n° 11 - 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
- ☞ calculer la complexité d'un algorithme glouton

A Occupation d'une 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 **strictement** supérieure à l'heure de fin de l'autre. **On cherche à maximiser le nombre de spectacles programmés dans la salle** mais pas forcément le temps d'occupation de la salle.

On dispose de trois heuristiques :

H1 choisir les spectacles compatibles les plus courts,

H2 choisir les spectacles compatibles qui débutent le plus tôt,

H3 choisir les spectacles compatibles qui se terminent le plus tôt.

A1. Appliquer à la main cet algorithme glouton avec l'heuristique H1, H2 et H3 à la liste de spectacles `[(6, 8), (1, 10), (7, 10), (3, 6)]`.

> **Solution :** On trouve :
>
> H1 `[(6, 8)]`
>
> H2 `[(1, 10)]`
>
> H3 `[(3, 6), (7, 10)]`

A2. Écrire une fonction gloutonne pour planifier ces spectacles dont la signature est `planify(L: list[tuple])-> list`, où `L` est la liste des spectacles triée dans l'ordre de l'heuristique choisie. Cette fonction renvoie la liste des spectacles planifiés représentés par leur tuple.

A3. Tester les trois heuristiques de la manière suivante :

```
spectacles = [(6, 8), (1, 10), (7, 10), (3, 6)]
print(spectacles)
print("H1",planify(sorted(spectacles, key = lambda x: x[1]-x[0])))
print("H2",planify(sorted(spectacles, key = lambda x: x[0])))
print("H3",planify(sorted(spectacles, key = lambda x: x[1])))
# La syntaxe lambda est hors programme, ne pas l'apprendre.
```

Solution :

```
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
if __name__ == '__main__':
    spectacles = [
        (6, 8),  # Durée 2 (Au milieu)
        (1, 10),  # Durée 9 (Englobe tout, commence très tôt)
        (7, 10),  # Durée 3 (Chevauche (6,8))
        (3, 6)  # Durée 3 (Chevauche (6,8))
    ]
    print(spectacles)
    print("H1", planify(sorted(spectacles, key=lambda x: x[1] - x[0])))
    print("H2", planify(sorted(spectacles, key=lambda x: x[0])))
    print("H3", planify(sorted(spectacles, key=lambda x: x[1])))
```

B Remplir son sac à dos

On cherche à remplir un sac à dos avec des objets. Chaque objet est **unique** et **insécable**[1] et possède une valeur et un poids connu. On cherche à maximiser la valeur totale emportée dans le sac à dos tout en limitant[2] le poids à `pmax`.

On dispose de plusieurs objets de valeur et de poids modélisés par une liste de tuples
`objets=[(60, 10), (100, 20), (120, 30), (54, 9)]`
non ordonnée.

`objets[i][0]` désigne la valeur de l'objet i et `objets[i][1]` son poids. On peut déconstruire un tuple par la syntaxe `v,p = objets[i]`

B1. Coder une fonction gloutonne et itérative pour résoudre ce problème. L'heuristique utilisée est celle de la **valeur maximale compatible en premier**. Sa signature est `greedy_kp(objets: list[tuple], pmax: int)-> list[tuple]`. Elle renvoie la liste des objets introduits dans le sac représentés par le

1. Soit on le met dans le sac, soit on ne le met pas. Mais on ne peut pas le diviser en plusieurs parties.
2. On accepte un poids total inférieur ou égal à `pmax`.

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`.

Solution :

```
def greedy_kp(objets, Pmax):
    Ptotal = 0
    Vtotale = 0
    sac = []
    objets = sorted(objets, reverse=True)  # Max val en premier
    i = 0
    while i < len(objets) and Ptotal < Pmax:
        v, p = objets[i]  # Max val en premier
        if Ptotal + p <= Pmax:  # solution ?
            sac.append((v, p))  # mettre dans le sac
            Ptotal += p
            Vtotale += v
        i += 1  # continuer
    return sac, Vtotale, Ptotal


def greedy_kp_for(objets, max_weight):
    Ptotal = 0
    Vtotale = 0
    sac = []
    objets = sorted(objets, reverse=True)  # Max val en premier
    for v, p in objets:
        if Ptotal + p <= max_weight:  # solution ?
            sac.append((v, p))  # mettre dans le sac
            Ptotal += p
            Vtotale += v
    return sac, Vtotale, Ptotal
```

B2. On choisir maintenant l'heuristique du **ratio** valeur sur poids maximal compatible en premier. Coder une fonction gloutonne et itérative qui implémente cette stratégie. Sa signature est `ratios_greedy_kp(objets, pmax)-> list[tuple]`.

Solution :

```
def ratios_greedy_kp(objets, Pmax):
    Ptotal = 0
    Vtotale = 0
    sac = []
    ratios = sorted([(v / w, v, w) for v, w in objets], reverse=True)
    i = 0
    while i < len(ratios) and Ptotal < Pmax:
        r, v, p = ratios[i]  # choix glouton !
        if Ptotal + p <= Pmax:  # solution ?
            sac.append((v, p))  # mettre dans le sac
            Ptotal += p
            Vtotale += v
```

```
            i += 1  # continuer
        return sac, Vtotale, Ptotal


    def ratios_greedy_kp_for(objets, Pmax):
        Ptotal = 0
        Vtotale = 0
        sac = []
        ratios = sorted([(v / w, v, w) for v, w in objets], reverse=True)
        for r, v, p in ratios:  # choix glouton !
            if Ptotal + p <= Pmax:  # solution ?
                sac.append((v, p))  # mettre dans le sac
                Ptotal += p
                Vtotale += v
        return sac, Vtotale, Ptotal
```

B3. Comparer les deux stratégies précédentes sur la liste d'objets `[(60, 10), (100, 20), (120, 30), (54, 9)]` à l'aide du programme principal suivant et conclure.

```
if __name__ == '__main__':
    # Format : (valeur, poids)
    objets = [(60, 10), (100, 20), (120, 30), (54, 9)]
    pmax = 30
    print(objets, pmax)
    gkp = greedy_kp(objets, pmax)
    print(gkp)
    rgkp = ratios_greedy_kp(objets, pmax)
    print(rgkp)
```

Solution : On trouve :

valeur `([(120, 30)], 120, 30)`

ratios `([(60, 10), (54, 9)], 114, 19)`

Aucune de ces deux stratégies n'est donc optimale puisqu'on pouvait prendre `[(100,20),(60,10)]`.

C Rendre la monnaie

Un commerçant doit rendre la monnaie à un client en minimisant le nombre de pièces utilisées. La somme d'argent à rendre est une somme entière m. On considère qu'il dispose d'autant de pièces et de billets qu'il le souhaite parmi le système monétaire euro.

C1. En utilisant un algorithme glouton, coder une fonction itérative dont la signature est :
`ccp(m: int, V: list[int])-> list[tuple]`
où `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)]`.

Solution :

```
def ccp(m, V):
    a_rendre = m
    solution = []
    for v in V:  # choix glouton
        n = a_rendre // v  # combien de fois ?
        if n > 0:  # on peut l'utiliser
            solution.append((n, v))  # prendre
            a_rendre = a_rendre % v
    if a_rendre == 0:
        return solution
    else:
        return None  # pas de solution
```

C2. 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 ?

Solution : Oui, cela semble optimal.

C3. Coder une fonction récursive `rec_ccp(m, V)` équivalente à la fonction précédente.

Solution :

```
def rec_ccp(m, V):
    if len(V) == 0 or m == 0:  # condition d'arrêt
        return []
    else:
        v = V[0]  # plus grande valeur
        n = m // v  # combien de fois ?
        if n > 0:  # on peut l'utiliser
            return [(n, v)] + rec_ccp(m - n * v, V[1:])
        else:
            return rec_ccp(m, V[1:])
```

C4. 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 :

```
V = [30, 24, 12, 6, 3, 1]
change = 49
print(change, ccp(change, V))
#[(1, 30), (1, 12), (1, 6), (1, 1)] pas optimal
# 4 pièces au lieu de 3 -->  2 x 24 +1
```

D Découper d'une barre de métal

On considère une barre de métal de longueur entière. La vente à la découpe de cette barre procure des revenus différents selon la longueur des découpes. 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 = [(1, 1), (14, 4), (20, 5)]` 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 la barre en maximisant la valeur qui en résulte d'après le calcul rapport prix/longueur. Cette fonction a pour signature : `greedy_cut(V: list[tuple], total_length: int)-> list[tuple]`.

Solution :

```
def greedy_cut(V, longueur):
    ratios = sorted([(v / l, v, l) for v, l in V], reverse=True)
    print(ratios)
    restant = longueur
    Prixtotal = 0
    S = []  # solution
    i = 0
    while i < len(ratios) and restant > 0:
        ratio, p, l = ratios[i]  # choix glouton
        n = restant // l  # combien de fois ?
        if n > 0 and l <= restant:  # solution ?
            S.append((p, l, n))
            Prixtotal += n * p
            restant -= n * l
        i += 1
    return S, restant, Prixtotal
```

D2. Tester la fonction à l'aide de ce programme principal. Conclure.

```
if __name__ == '__main__':
    decoupes = [(1, 1), (14, 4), (20, 5)]
    print(decoupes)
    longueur = 8
    print("Longueur -> ", longueur)
    gc = glouton(decoupes, longueur)
    print(gc)
```

Solution : L'algorithme glouton n'est pas optimal ! Il trouve 23 alors qu'on peut obtenir 24 !

R Cet exercice est un problème qui présente des caractéristiques du sac à dos et du rendu de monnaie. Le code est très proche dans sa structure. C'est tout l'intérêt de la description algorithmique des problèmes : généraliser les résolutions.

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))]
```

Solution : On choisit de placer les cours dans la première salle disponible, à partir de la salle 0. Si cela n'est pas possible on cherche dans la salle suivant et ainsi de suite. Si aucune salle n'est disponible, on en crée une.

E2. Écrire une fonction gloutonne de signature `allouer_les_salles(cours)` 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…

Solution :

```
def allouer_les_salles(cours):
    cours = sorted(cours, key=lambda tup: tup[1][0], reverse=True)
    # print(lectures)
    planning = []
    while len(cours) > 0:
        titre, (debut, fin) = cours.pop()  # choix glouton
        salle = 0
        positionne = False
        while salle < len(planning) and not positionne:  # solution ?
            if debut >= planning[salle][-1][1][1]:
                planning[salle].append((titre, (debut, fin)))
                positionne = True
            else:
                salle += 1  # salle suivante
        if not positionne:  # échec
            planning.append([])  # créer une nouvelle salle
            planning[-1].append((titre, (debut, fin)))
    return planning
```

E3. Que pensez-vous du nombre de salles obtenu par l'algorithme ?

Solution : A priori, le nombre de salles obtenu par l'algorithme est minimal si les horaires des cours sont figés. Maintenant, si on pouvait déplacer les cours…