CentraleSupélec

Algorithmique et complexité

TD 7/7 – Résolution de problèmes NP-difficiles

L'objectif de ce TD est de traiter un problème NP-difficile.

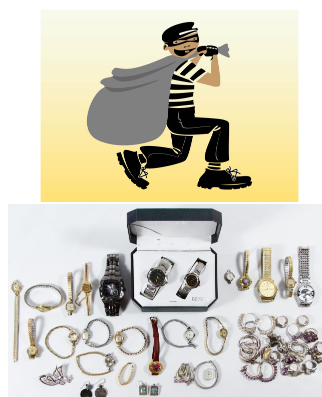
Exercice 1 : Problème du Sac à dos

Considérons trois différents problèmes pratiques.

Bagage de cabine

Vous vivez une vie nomade, voyageant tout le temps autour du monde. Vous avez un ensemble d'éléments auxquels vous tenez plus ou moins selon vos besoins, par exemple votre ordinateur portable est quelque chose de très important, il a donc une grande valeur. Au contraire, vous aimez votre t-shirt SF, mais vous pouvez toujours en acheter un à destination, alors votre t-shirt SF a une petite valeur. Vous ne transportez que des bagages de cabine. Différentes compagnies aériennes imposent des limites différentes au poids des bagages à main, par exemple 7kg ou 10kg. Comment maximiser la valeur des articles que vous pouvez embarquer avec vous, tout en restant en dessous du poids maximum autorisé ?

Cambriolage



Un voleur parvient à pénétrer chez un bijoutier. Il y a beaucoup d'objets qu'il peut voler : montres, bagues, colliers ... Tous ces éléments ont des valeurs et des poids différents. Le voleur ne peut transporter qu'un poids limité. Supposons que les articles peuvent être divisés en catégories, chaque catégorie ayant un poids, une valeur et un certain nombre d'éléments disponibles. Comment le voleur devrait-il choisir quoi prendre ?

Repas équilibré

Vous recevez une liste de valeurs nutritionnelles pour une portion de chaque élément servi dans le menu quotidien de votre restaurant universitaire préféré. Pour chaque élément, vous obtenez le nombre de calories (ou poids) et la valeur protéique d'une partie de l'article. Qu'est-ce que vous avez choisi de manger pour maximiser la prise de protéines sans dépasser une certaine quantité de calories ?

Complexité

Question 1

Les trois problèmes peuvent être formalisés en utilisant le même format. Donnez une description formelle de ce problème : quelles sont les entrées, quelle est la question du problème de décision correspondant, quelle est la question du problème d'optimisation ?

Élements de correction :

Tous ces problèmes sont des problèmes de sac à dos (Knapsack).

Le problème de décision :

KNAPSACK - DECISION
Entrée :
- O un ensemble de n objets $o_1, \ldots, o_n$, chaque o_i a un poids w_i et une valeur v_i
- W une limite maximum de poids
- V une valeur souhaitée

Question : Y a t il un sous-ensemble $S \subseteq \{1, \ldots, n\}$? tel que :
- $\sum_{i \in S} w_i \leq W$
- $\sum_{i \in S} v_i \geq V$

La version optimisation correspondante :

KNAPSACK - OPTIMIZATION
Entrée :
- O un ensemble de n objets $o_1, \ldots, o_n$, chaque o_i a un poids w_i et une valeur v_i
- W une limite maximum de poids

Question : Quel est le sous-ensemble $S \subseteq \{1, \ldots, n\}$ ayant la valeur maximum $\sum_{i \in S} v_i$? tel que :
- $\sum_{i \in S} w_i \leq W$

Pour simplifier, vous pouvez supposer que toutes les variables sont entières.

Question 2

Montrez que ce problème (connu comme **KNAPSACK problem**) est NP-complet. Vous pouvez utiliser un problème NP-complet classique SUBSET-SUM :

SUBSET SUM
Entrée :
- un ensemble A d'entiers positifs $a_1, \ldots, a_n$
- une valeur $t \in \mathbb{N}$

Question : Existe-il un sous-ensemble $S \subseteq [1, n]$ dont la somme satisfait $\sum_{i \in S} a_i = t$?

Élements de correction :

La NP-complétude est étudiée sur le **problème de décision**.

Première étape, KNAPSACK est NP car nous pouvons écrire **un algorithme polynomial (linéaire) pour vérifier une solution** (nous avons besoin de calculer la somme des valeurs $\sum_{i \in S} v_i$ et la somme des poids $\sum_{i \in S} w_i$ de la solution proposée $S \subseteq \{1, \ldots, n\}$).

Ensuite, nous devons **réduire SUBSET SUM à KNAPSACK**. Étant donné une instance de SUBSET SUM comme ci-dessus, nous construisons (**en temps polynomial**) l'instance du problème de sac à dos suivante :
- O un ensemble de n objets $o_1, \ldots, o_n$ où $w_i = v_i = a_i$ (1)
- $W = V = t$ (2)

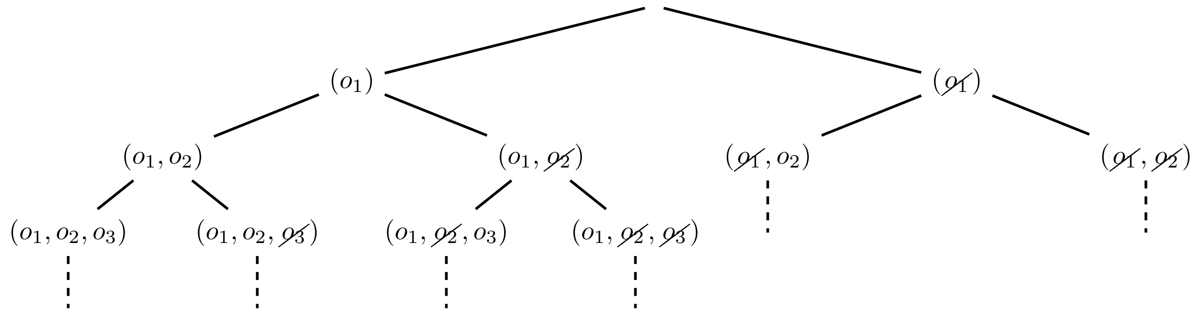
Nous devons vérifier que S est une solution de l'instance SUBSET SUM **si et seulement si** S est une solution du problème KNAPSACK ciblé. Ce qui est évident puisque (1) et (2) nous donnent :

$$\sum_{i \in S} w_i \leq W \wedge \sum_{i \in S} v_i \geq V \Longleftrightarrow \sum_{i \in S} a_i = t$$

.

Backtracking

On considère l'algorithme de **backtracking** qui décide à chaque étape la présence ou l'absence d'un objet dans la solution partielle courante. On construit ainsi l'espace de solutions qui prend la forme d'un arbre binaire de hauteur n :



L'algorithme de *backtracking* va énumérer toutes les solutions possibles pour trouver la meilleure (ici ayant la plus grande valeur).

L'algorithme de *backtracking* évitera d'explorer une branche si elle dépasse déjà la limite de poids.

Question 3

Écrivez en Python le code de cet algorithme. Pour vous aider, rendez-vous sur la page de pratique du TD :

https://wdi.centralesupelec.fr/1CC2000/TD7ProgEn

Élements de correction :

```
# computes the children of the current partial solution
def children(curParSol):

    childrenSols = []
    ############### TODO : complete code ####################
    if curParSol['index'] < len(O_dict):

        first_child = curParSol.copy() # It is not a deep copy ('selected' is not copied)
        first_child['index'] += 1
        childrenSols.append(first_child)

        next_obj = objs_list[curParSol['index']]
        if curParSol['weight'] + O_dict[next_obj]['w'] <= W:
            second_child = {'selected': curParSol['selected'] | {next_obj},  # union
                            'index':    curParSol['index']    + 1,
                            'weight':   curParSol['weight']   + O_dict[next_obj]['w'],
                            'score':    curParSol['score']    + O_dict[next_obj]['v']}
            childrenSols.append(second_child)

    return childrenSols
```

Glouton

Un algorithme **glouton** construira une solution pas à pas sans remettre en question les choix effectués. La solution obtenue n'aura pas de garantie d'optimalité.

Question 4

Proposez un algorithme **glouton** le plus efficace possible pour le problème du sac à dos.

Élements de correction :

On commence par trier les éléments dans l'ordre décroissant selon leur rapport $\frac{valeur}{poids}$.

Puis on considère chaque élément dans l'ordre et on l'ajoute à la solution si l'élément rentre dans le sac.

Remarque : attention de ne pas s'arrêter au premier élément qui ne rentre pas.

Question 5

Quelle est la complexité en temps de votre algorithme ?

Élements de correction :

La partie de l'algorithme la plus couteuse en temps est le tri de la liste.

La complexité en temps est donc de $\mathcal{O}(n * log(n))$.

Question 6

Quelle serait la pire instance pour cet algorithme ?

Élements de correction :

Soit B la capacité du sac à dos et considérons $n = 2$ objets o_1 et o_2 tel que $w_1 = B$ et $v_1 = B - 1$ avec $w_2 = v_2 = 1$. L'algorithme glouton choisira o_2 et s'arrêtera (faute d'espace pour mettre o_1), cependant la solution optimale consiste à prendre l'objet o_1.

La solution obtenue par l'algorithme glouton est $(B - 1)$ pire que la solution optimale. On dit que l'algorithme glouton est *arbitrairement* mauvais (pour toute valeur k, il est possible de créer une instance du problème tel que la solution approchée soit k fois pire que l'optimum).

Question 7

Retournez sur la page de pratique du TD. Complétez le code concernant l'algorithme glouton. Un *benchmark* est fourni pour comparer le temps d'exécution de *backtracking* et de *glouton* sur des instances aléatoires. Un autre *benchmark* est fourni pour comparer la qualité des solutions obtenues par ces algorithmes.

Élements de correction :

```
objs_list = sorted(O_dict.keys(), key = lambda obj: O_dict[obj]['v'] / O_dict[obj]['w'], \
        reverse=True)

for obj_name in objs_list:
    next_obj = O_dict[obj_name]
    if sol['weight'] + next_obj['w'] > W :
        continue
    sol['selected'].add(obj_name)
    sol['weight'] += next_obj['w']
    sol['score'] += next_obj['v']
```

Programmation dynamique

Nous voulons proposer un algorithme de programmation dynamique pour résoudre le problème du sac à dos. On note $V(i, j)$ la valeur totale maximale qu'on peut mettre dans un sac à dos de taille j en embarquant que des objets parmi $o_1, \ldots, o_i$.

Question 8

Quelle est la formule de récursion qui calcule $V(i, j)$ pour tout i et j dans $\mathbb{N}$?

Élements de correction :

We suppose here that all weights w_i and W are strictly positive natural numbers. The recursive formula is as follows :

- $V(0, j) = 0$
- $V(i, 0) = 0$
- $1 \leq i$ et $1 \leq j < w_i$ alors $V(i, j) = V(i - 1, j)$
- $1 \leq i$ et $w_i \leq j$ alors $V(i, j) = max\,(V(i - 1, j), v_i + V(i - 1, j - w_i))$

En effet, on peut soit obtenir la meilleure solution avec les $i - 1$ premiers objets mais sans utiliser l'objet o_i, soit trouver la meilleure solution en utilisant o_i conjointement avec la meilleure solution pour les $i - 1$ premiers objets avec un poids maximal réduit de w_i.

Question 9

Quelle est la complexité en temps et en espace d'un algorithme de programmation dynamique implémentant la formule précédente ?

Élements de correction :

Pour répondre au problème d'optimisation KNAPSACK, nous devons calculer $V(n, W)$.

Afin d'éviter les calculs redondants, nous utiliserons une matrice V de taille $(n + 1) \times (W + 1)$.

La complexité en temps et en espace est de $\mathcal{O}(n \times W)$.

Question 10

Cela signifie-t-il que $P = NP$?

Élements de correction :

A première vue, il semble que nous ayons résolu un problème NP-difficile en temps polynomial !

C'est un piège, il faut comprendre que les entrées sont des nombres donnés en représentation binaire. Seul un nombre logarithmique de bits est nécessaire pour écrire un nombre !

La complexité de l'algorithme est de $W \times n$ et elle n'est pas polynomiale dans la taille de l'entrée. Il faut un nombre exponentiel d'opérations $W = exp(log(W))$ par rapport à la taille de l'entrée $log(W)$.

Question 11

Retournez sur la page de pratique du TD. Écrivez en Python puis tester l'algorithme de résolution basé sur la programmation dynamique. Un *benchmark* est fourni pour comparer le temps d'exécution de tous les algorithmes.

Élements de correction :

```
def Knapsack_DP(O_dict, W):

    n = len(O_dict)
    objs_list = sorted(O_dict.keys())

    # FILLING THE TABLE ITERATIVELY
    V = np.zeros((n+1, W+1), dtype='int32')

    ############### TODO : complete code ####################
    for i in range(1,n+1):
        for c in range(1,W+1):
            obj = O_dict[objs_list[i-1]]
            if c >= obj['w']:
                V[i,c] = max(V[i-1,c], V[i-1, c-obj['w']] + obj['v'])
            else:
                V[i,c] = V[i-1,c]

    # RETRIEVE THE SOLUTION
    selected = set()

    ############### TODO : complete code ####################
    i, w = n , W
    while i != 0:
        if V[i,w] != V[i-1,w]:
            obj = objs_list[i-1]
            selected.add(objs_list[i-1])
            w -= O_dict[obj]['w']
        i -= 1

    return {'selected': selected, 'score': V[n,W]}
```