Algorithmics and complexity

TD 7/7 – Resolution of NP-hard Problems

The goal of this TD is to work on solving NP-hard problems.

Exercise 1 : Knapsack problem

Let us consider three different practical problems.

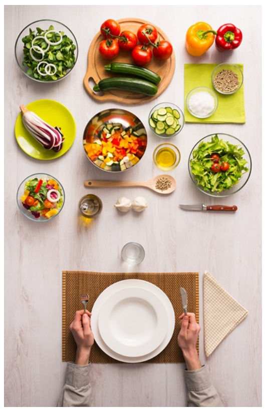
Cabbin Luggage

You live a nomad life, always traveling around the world. You have a set of items which you value differently, for exemple your laptop is something you always want with you hence has a high value. On the contrary, you like your SF t-shirt, but you can always buy a t-shirt at destination, so your SF t-shirt has a small value. You only carry cabin baggage. Different airlines put different limits on the weight of cabin baggage, for example 7kg or 10kg. How do you maximize the value of items you can bring with you, while remaining below the maxium weight allowed ?

Robbery

A thief manages to break into a jewelry. There are many items he can rob : watches, rings, necklaces... All these items have different values and different weights. The thief can only carry a limited weight of goods. Assume items can be split into categories, each category having a weight, a value and a number of items occuring in this category. How should the thief chose what to take ?

Balanced Lunch



You are given a list of nutritional values for a portion of each item served in the daily menu of your favorite university restaurant. For each item, you get the number of calories (or weight) and the protein value of a portion of the item. What do you chose to eat to maximize the proteins you get and without overtaking a certain amount of calories ?

Complexity

Question 1

All three problems can be formalized using the same framework. Give a formal description of this problem : what are the inputs, what is the decision question, what is the optimization question ?

Solution elements :

All these problems are knapsack problems.

The binary decision problem is :

KNAPSACK - DECISION
Inputs :
- O a set of n objects $o_1, \dots, o_n$, each o_i has a weight w_i and a value v_i
- W a maximum Weight limit
- V the expected value

Question : Is there a subset $S \subseteq \{1, \dots, n\}$? such that :
- $\sum_{i \in S} w_i \leq W$
- $\sum_{i \in S} v_i \geq V$

And the associated optimization problem is :

KNAPSACK - OPTIMIZATION
Inputs :
- O a set of n objects $o_1, \dots, o_n$, each o_i has a weight w_i and a value v_i
- W a weight limit

Question : What is the subset $S \subseteq \{1, \dots, n\}$ having the maximum value for $\sum_{i \in S} v_i$? such that :
- $\sum_{i \in S} w_i \leq W$

For simplification, you can assume that all variables are integers.

Question 2

Show that this problem, known as **KNAPSACK problem**, is NP-complete. You can use the SUBSET-SUM problem which is known to be NP-complete :

SUBSET SUM
Inputs :
- a set A of non-negative integers $a_1, \dots, a_n$
- a value $t \in \mathbb{N}$

Question : Is there a subset $S \subseteq [1, n]$ with a total sum $\sum_{i \in S} a_i = t$?

Solution elements :

The NP-completeness is studied on the **decision problem**.

First, KNAPSACK is NP because we can write **a polynomial (linear) algorithm to check a solution** (we only need to compute the sum of values $\sum_{i \in S} v_i$ and the sum of weights $\sum_{i \in S} w_i$ of the proposed solution $S \subseteq \{1, \dots, n\}$).

Now, we must **reduce SUBSET SUM to KNAPSACK**. Given a subset-sum instance as above, we build (**polynomially**) the following knapsack instance :
- O a set of n objects $o_1, \dots, o_n$ where $w_i = v_i = a_i$ (1)
- $W = V = t$ (2)

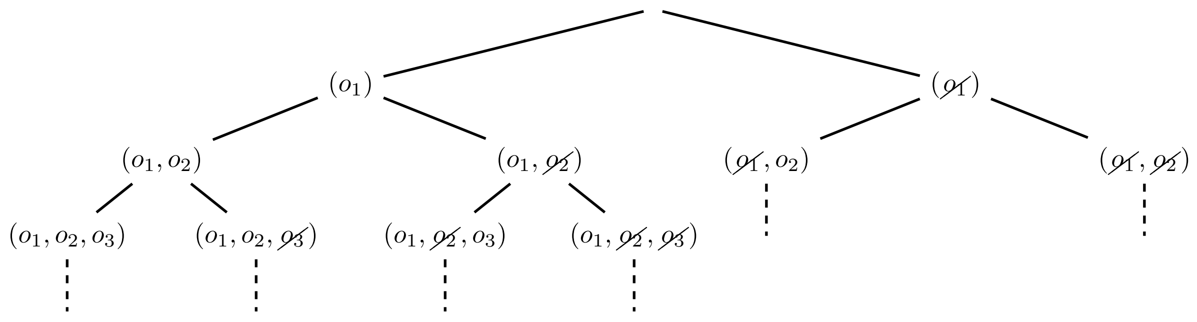
We need to verify that S is a solution of the SUBSET SUM instance **if and only if** S is a solution of the targeted KNAPSACK instance. Which is obvious since under (1) and (2) we have :

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

.

Backtracking

We Consider a **backtracking** algorithm that decides the presence or absence of an item in the solution (under construction) at each step. We obtain the following branching binary tree of height n :



(o_1)
$(\cancel{o_1})$
(o_1, o_2)
$(o_1, \cancel{o_2})$
$(\cancel{o_1}, o_2)$
$(\cancel{o_1}, \cancel{o_2})$
(o_1, o_2, o_3)
$(o_1, o_2, \cancel{o_3})$
$(o_1, \cancel{o_2}, o_3)$
$(o_1, \cancel{o_2}, \cancel{o_3})$

The backtracking algorithm will enumerate all possible solutions to find the one having the best score (the greatest value here).

The backtracking algorithm will avoid to explore a branch if it already exceed the weight limit.

Question 3

Write the Python code of this algorithm. To help you, go to the practice page of the 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
```

Greedy

A **greedy** algorithm constructs a solution step by step without going back on decisions already taken. There is no guaranty of optimality for the solution.

Question 4

Propose a **greedy** algorithm as efficient as possible for the knapsack problem.

Solution elements :

We start by ordering the elements in a decreasing order based on the ratio $\frac{value}{wheight}$.

Then we consider each element in order and we add it to the current solution if it fits in the bag.

Remark : be careful not to stop after the first element that does not fit.

Question 5

What is the time complexity of this algorithm ?

Solution elements :

The most time consuming part of the algorithm is the sorting of the list.

The time complexity it therefore $\mathcal{O}(n * log(n))$.

Question 6

What would be the worst instance for this algorithm ?

Solution elements :

Let B be the capacity of the bag. We consider $n = 2$ objects o_1 and o_2 such that $w_1 = B$ and $v_1 = B - 1$ with $w_2 = v_2 = 1$.

The greedy algorithm choose o_2 and the stop (no more space for o_1). However, the optimal solution consists in selecting o_1.

The solution obtained by the greedy algorithm is $(B - 1)$ times worst than the optimal solution.

We say that the greedy algorithm is arbitrarily bad (for every value k, it is possible to create an instance of the problem such that the obtained solution is k times worst than the optimal solution).

Question 7

Go back to the practice page of the TD. Complete the code concerning the greedy algorithm.

A *benchmark* is given in order to compare the execution time of *backtracking* and *greedy* on random instances. An other *benchmark* is given to compare the quality of the solutions obtained by those algorithms.

Solution elements :

```
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']
```

Dynamic Programming

We want to propose a dynamic programming algorithm to solve the knapsack problem. We note $V(i, j)$ the maximum total value that can be inserted into the knapsack of capacity j picking up objects among $o_1, \ldots, o_i$.

Question 8

What is the recursion formula that computes $V(i, j)$ for all i and j in $\mathbb{N}$?

Solution elements :

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

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

Indeed, you can either obtain the best solution with the $i-1$ first objects but without using the object o_i, or find the best solution using o_i together with the best solution for the $i-1$ first objects with maximum weight reduced by w_i.

Question 9

What would be the time and the space complexity of a dynamic programming algorithm implementing the formula above ?

Solution elements :

To answer the KNAPSACK optimization problem we need to compute $V(n, W)$.

In order to avoid repeating computations, we will use a matrix V of size $(n+1) \times (W+1)$

Both time and space complexity are in $\mathcal{O}(n \times W)$.

Question 10

Does this means that $P = NP$?

Solution elements :

At first sight, it seems that we solved an NP-hard problem in polynomial time ! !

This is tricky, you need to understand that the inputs are numbers given in binary representation. Only a logarithmic number of bits is required to write a number !

The complexity of the algorithm is $W \times n$ and it is not polynomial in the size of the entry. You need an exponential number of operations $W = exp(log(W))$ wrt the size of the entry $log(W)$.

Question 11

Back to the practice page of the TD, write and test this dynamic programming algorithm.

A *benchmark* is given to compare the execution time between all the algorithms.

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]}
```