CentraleSupélec

Algorithmics and complexity

TD 6/7 – Packing Problem

The goal of this TD is to compare different heuristics of the same family to solve/approximate an NP-complete problem.

Exercise 1 : Problem Analysis

A company provides *Cloud Computing* services to its customers. This company has powerful machines available upon reservation to perform computations that are CPU-intensive. Every customer can execute its tasks after reserving some amount of computation time. Each task will be processed on a single machine.

The company has M servers available B minutes a day. It may need to rent additional servers but wishes, as far as possible, to avoid this option. Every night, it must affect each of the N customers tasks to a machine. The tasks are then executed the next day.

The company wants to design a software to know if it is possible to assign each task on a machine without the need to rent additional servers.

Question 1

Give a formal definition of this problem (give the entries and the question of the problem).

Élements de correction :

The problem we want to solve is the Bin Packing.

BIN PACKING
Inputs :

- a set O of N objects of size $o_i \in \mathbb{N}$
- $B \in \mathbb{N}$ a bag size
- M a number of available bags

Question : Is there an assignment of the N objects in the M bags, such that the sum of sizes of the items in each bag is less than or equal to B

Formally, we can represent the assignment in several ways :

- a function f which associates to each object number a bag number ($f : [1, N] \to [1, M]$) verifying : $\forall j \in [1, M]. \ \sum_{i \in f^{-1}(\{j\})} o_i \leq B$;
- a partition of the objects into M subsets;
- ...

Question 2

Show that the previously formalized problem is *NP*-Complete. You can use the *NP*-Complete Partition problem:

> **PARTITION**
> **Inputs** : Let E a set of natural numbers.
> **Question** : is there a partition of E into two sets E_1 and E_2 such that:
>
> $$\sum_{e_i \in E_1} e_i = \sum_{e_j \in E_2} e_j$$

Élements de correction :

We start by showing that the problem is in *NP*. We give an algorithm which checks a solution in polynomial time.

In Python and in $\mathcal{O}(n \log n)$:

```
O = [o1, o2, ..., oN]
M =
B =

Aff = bin_packing()
>>> Aff is a list of lists : Aff is a list of bags,
    a bag is a list of objects, an object is an integer

def verify(O,Aff):
    if len(Aff) > M: # M bags are enough
        return False
    Aff_flattened = [val for bin in Aff for val in bin]
    if sorted(O) != sorted(Aff_flattened): # it is indeed an assignment
        return False
    for bin in Aff: # no bag is overflowing
        if sum(bin) > B:
            return False
    return True
```

Then we show that we can reduce an NP-complete problem (here Partition) to our Bin Packing problem. From an instance of Partition $\mathcal{I}_{Partition} = \langle E \rangle$, we build the instance $\mathcal{I}_{BinPacking} = \langle O, M, B \rangle$ of the Bin Packing problem as follows:

- $O = E$
- M =2
- $B = \frac{1}{2} \times \sum_{e \in E} e$

It remains to show that: $\mathcal{I}_{Partition}$ is a positive instance if and only if $\mathcal{I}_{BinPacking}$ is also a positive instance

Question 3

Define the corresponding optimization problem (give the entry and the question of the problem).

Élements de correction :

To formalize an optimization problem, we need to specify a function *score* to optimize (minimize or maximize):

BIN PACKING
Inputs :

- a set O of N objects of size $o_i \in \mathbb{N}$
- $B \in \mathbb{N}$ a bag size

Question : find an assignment ($f : [1, N] \to \mathbb{N}$ verifying $\forall j \in img(f).\ \sum_{i \in f^{-1}(\{j\})} o_i \leq B$) minimizing the number of bags i.e. $score(f) = card(img(f))$ where $img(f)$ be the image set of f.

Question 4

Why do we think that there is no polynomial time algorithm to solve this problem?

Élements de correction :

If we solve the optimization problem in polynomial time, we answer the decision problem in polynomial time too. It is sufficient to compare the optimal number of bags M^* at the output of the optimization problem with the number of bags M at the input of the decision problem. We answer "yes" to the decision problem if and only if $M \geq M^*$.

Finding a polynomial algorithm to solve an optimization problem associated with an NP-complete decision problem is impossible under the postulate of $\mathcal{P} \neq \mathcal{NP}$.

Exercice 2 : Resolution algorithms

We want to solve the Bin-Packing problem formalized above:

Question 1

Propose a greedy algorithm to solve the problem. Are there many?

Question 2

Write the algorithm (in pseudo-code or in Python) and give its complexity.

Question 3

Run your algorithm on an instance with a bin of size 10 and the following items:

$$4, 4, 5, 5, 5, 4, 4, 6, 6, 2, 2, 3, 3, 7, 7, 2, 2, 5, 5, 8, 8, 4, 4, 5$$

Élements de correction :

We build the solution step by step without ever questioning/changing a choice that has been made: once an object has been added to a bag, we will never change it. For this allocation problem, we can consider many strategies: First Fit, Best Fit, Next Fit...

First Fit (FF) : add the objects, one after the other, in the first possible bag (the one with the smallest number and in which there is enough space). If there is no bag with eenough space, open a new one.

```
def FirstFit(O, B):
    Aff = []
    for o in O:
        for bin in Aff:
            if sum(bin) + o <= B :
                bin.append(o) # if we find a bag where it fits
                break
        else:
            Aff.append([o]) # otherwise we create a new bag

    return Aff
```

BestFit (BF) : we put an object in the bag which is the most filled and in which there is enough space ;

```
def BestFit(O,B):
    Aff = []
    for o in O:
        idx_bin = None
        min_space = B
        for idx in range(len(Aff)):
            space = B - sum(Aff[idx])
            # if it has enough space for the object, it becomes the new best fit
            if o <= space and space < min_space :
                idx_bin = idx
                min_space = space
        if idx_bin == None:
            # If we found no bin for our object, use a new bin
            Aff.append([o])
        else:
            Aff[idx_bin].append(o)

    return Aff
```

The complexity of the FF and BF algorithms is $\mathcal{O}(N^2)$. Indeed, the outer loop is executed exactly N times and, in the worst case, we will use/run the N bags (inner loop).

The two algorithms above can be used/improved by having previously sorted the objects by decreasing order of size. We obtain the algorithms *First Fit Descreasing (FFD)* and *Best Fit Decreasing (BFD)*.

The worst-case complexity of these algorithms does not change because the cost of a sort $\mathcal{O}(N \log(N))$ is less than the complexity of the rest $\mathcal{O}(N^2)$.

Élements de correction :

Approach in $\mathcal{O}(N)$:

NextFit (NF) : we put the element in the last opened bag when there is enough space left ; otherwise a new bag is opened.

```
def NextFit(O,B):
    Aff = [[]]
    for o in O:
        # check if there is space left in the last bag
        if sum(Aff[-1]) + o > B :
            Aff.append([o]) # open a new bag and add the object
        else :
            Aff[-1].append(o) # add the object to the last bag

    return Aff
```

Results:

- NF: 14 bags
- FF: 13
- FFD: 11! optimal (because the sum of the objects is equal to 110)
- BF: 12
- BFD: 11! optimal

For example, an instance for which the BFD and FFD do not find the optimal solution. Bag size 13, and the elements (optimally grouped in 2 bags) : [3,5,5] [2,2,2,7].

In the TD-Practice, there is also an example where BFD and FFD do not give the same score.

Question 4

Estimate the efficiency of your algorithm : in the worst case, how far is your approximate solution from the minimal number of servers you need?

Élements de correction :

First it is convenient to assume that each bag can contain the weight 1. The weight of the objects is normalized with reference to the capacity of the bags: $p_i \longleftarrow \frac{o_i}{B}$.

What is the minimum number of bags we need ?

The answer is immediate, $\lceil \sum_{i=1}^{n} p_i \rceil$. We can never have less than $\lceil \sum_{i=1}^{n} p_i \rceil$ bags. This answer is true for any algorithm, not only FF.

What is a maximum number of bags we will use ?

Observation: there is at most one non-empty bag in which the available space is greater than or equal to $\frac{1}{2}$.

Proof: Suppose we have two bags i, j, $i < j$ that are more than half empty. We see that the items in the bag j could have been placed in bag i (whose number is lower) and FF should have put them in bag i. Our observation is then true.

So we have all the bags (except, perhaps, one) at least half full. Therefore, we never need more than $\lceil 2 \sum_{i=1}^{n} p_i \rceil$ bags.

Conclusion: We are sure that a solution with FF never exceeds the optimal solution twice: $k^* \leq k < 2k^*$ where k is our solution and k^* is the optimal solution.

For information, FF and BF are 17/10-approxs and BFD and FFD are 11/9-approxs