LINMA2450 - Homework 1: Integer Knapsack Problem

Gengler, Arthur
NOMA:1624 1700

Do, Dinh Thanh Phong
NOMA:1370 1700

3 november 2021

1 Algorithms explanation

As a reminder, the integer knapsack problem can be written with the following notation:

$$z = \max_x \sum_{i=1}^{N} x_i c_i \qquad \text{st.} \quad \sum_{i=1}^{N} x_i a_i \leq b, \quad x \in \mathbb{Z}_+^N$$

where the vector a_i,c_i represent respectively the weight and profit for the object x_i.

1.1 Greedy algorithm

Description of the algorithm

In order to perform the greedy algorithm, we first calculate the utility of each object defined as the ratio between the satisfaction (c_i) and the weight (a_i) then we create a list of size N with the object sorted according to their utility in decreasing order. We also create a dictionary of size N which will be used to count the number of times an object is added in the knapsack.
The next step is to add objects in the knapsack, to do so we go through the list (starting with the element with highest utility) and we look if we can add the object with index i in the knapsack if yes then we increase the total weight by the weight of the object and we start again, if not we pass to the object of index i+1.

Complexity

There are 2 main step. The first step is to apply a sort algorithm to our utility list. Such algorithm implemented by julia by default have a complexity of $\mathcal{O}\big(N\log(N)\big)$.
For the second step, we go through the objects list sorted previously and we add the maximum possible amount of an object without breaking our constrain on the weight. This step can be done in $\mathcal{O}(N)$.

Thus, the final complexity is:

$$\mathcal{O}\big(N\log(N)\big) + \mathcal{O}(N) = \big(N\log(N)\big)$$

1.2 Dynamic programming

Description of the algorithm

The idea is to divide the general knapsack problem into subproblem. So first we are going to consider the knapsack problem with 1 object then we will use this to for the knapsack problem for 2 objects and so on to finally solve the knapsack problem with N objects. First we construct a matrix, named table, of size (N+1, b+1) and we complete the matrix line by line, the line i corresponds to the case where we limit the knapsack problem to the first i objects. Then for each row j with limit ourself to a weight j. So for the element at position (i,j) we take the maximum value between the value of c_i(t-1)+table(i-1, j-a_i(t-1)) and c_i t+table(i-1, j-a_i t) where c_i is the satisfation of object i, a_i the weigth of object i and t goes from 1 to j/a_i. We also check that for every t we don't exceed the weight and stop the iteration of t when it does to go to the element (i, j+1).

Complexity

This algorithm goes through a $(N+1)\times(b+1)$ matrix one time and at each iteration it update the value of this matrix.
For our implementation, for each element in our $(N+1)\times(b+1)$ matrix, it update at most t times the element. We know that this $t < b$
Thus, we consider the complexity of the algorithm as follow:

$$\mathcal{O}\big((N+1)(b+1)t)\big) \leq \mathcal{O}\big((N+1)(b+1)b)\big)$$

2 Application to small instance

After solving the knapsack integer problem on the file `small.json` we obtain results that are sumarized inside the table 1:

Solver	Solution	Objective value
Gurobi	$x^* = (2,0,3,0,0,0,0,0,0,0)$	55
Greedy	$x^* = (3,2,0,0,0,0,0,0,0,0)$	49
Dynamic Programming	$x^* = (2,0,3,0,0,0,0,0,0,0)$	55

Table 1: Solution and different objective value for the different implementation of the integer knapsack problem

Before analysing the table, by a quick look of the file `small.json`. We can see that $a_i = c_i$. Which means that all the utility $c_i/a_i = 1$ for all objects. Moreover, $b = 55$. According to this table we can do a few observation:

- First, we can see that both the dynamic programming and gurobi solver return the same objective value (55) which is optimal since $b = 55$ and each object's utiliy is equal to 1. This confirm the theory which suggest that dynamic programming should give the optimal value on this kind of problem.
- Secondly, we observe that greedy algorithm doesn't return the optimal solution but only a feasible solution. In our implementation, since all object's utility are equal, the sorting operation wont change the order of x. The implementation add first x_1 a maximum number of time (before disrecpecting the constrain on the weight), then it does the same with x_2,...x_N.

3 Results

Gurobi	small (n = 10)	medium (n = 100)	large1 (n = 1000)	large2 (n = 1000)
Objective value z^*	55	14 552	108 680	2 194 836
Run time	0.133 s	0.198 s	60 s	1.961 s
Greedy	small (n = 10)	medium (n = 100)	large1 (n = 1000)	large2 (n = 1000)
Objective value z^*	49	14 550	108 680	2 194 838
Run time	0.235 s	0.241 s	0.228 s	0.264 s
Dynamic programming	small (n = 10)	medium (n = 100)	large1 (n = 1000)	large2 (n = 1000)
Objective value z^*	55	14 552	108 680	2 194 836
Run time	0.232 s	0.416 s	99.632605s	727.648 s

Table 2: Results and time taken for different solvers and different data sets

As the result suggest, run time increase with the number of object n in our knapsack problem. We can also observe that time complexity of the greedy algorithm is way lower than the one of the dynamic programming. We also see that the file `large2` takes more time for Greedy and dynamic programming than the file `large1` despite for both $n = 1000$. In our Gurobi solver, we decided to stop after 60s. We obtain the optimal solution. however, if we let the solver continue, it take about 1h15'.