LINMA2450 - Combinatorial Optimization
Homework 1

d'Oultremont Augustin - 2239 1700 Lemaire Valentin - 1634 1700

November 3, 2021

1 Introduction

The goal of this assignment was to compute a solution for the integer knapsack problem (which is known to be NP-Hard). A mathematical formulation of the problem can be found below

$$z = \max \sum_{i=1}^{n} x_i c_i \qquad (1) \qquad \sum_{i=1}^{n} x_i a_i \leq b \qquad (2) \qquad x \in \mathbb{Z}_+^n \qquad (3)$$

Where z is the final value of the objective, the x_i are the counts of how many times item i is packed into the knapsack. c_i are the utilities of the items, a_i their weights and b the maximum weight allowed to be packed into the knapsack.

To solve this problem we used 3 different approaches. The first was to formulate the problem using the JuMP Julia package and solve it using the Gurobi Optimizer, the second was to create a greedy algorithm and finally the third was to use Dynamic Programming.

2 Additional algorithms

2.1 Greedy algorithm

To create a greedy solution for this problem, we computed for each of the items the ratio of its utility divided by its weight: $r_i = \frac{c_i}{a_i}$.

At each iteration of the algorithm, we chose the item not in the bag that had the highest ratio r_i and put as many of them in the knapsack as could fit without breaking the maximum weight constraint (2) nor the integer constraint (3). This was done until the item with the smallest ratio was considered. Algorithm 1 shows pseudocode for this.

Algorithm 1 Greedy Knapsack Algorithm

$total\_weight \leftarrow 0$
$total\_utility \leftarrow 0$
$x_i \leftarrow 0 \quad \forall\, i \in \{1...n\}$
for $i \in \{1...n\}$ **do**
  $r_i \leftarrow \frac{c_i}{a_i}$
end for
$idx \leftarrow argsort(r)$
for $i \in idx$ **do**
  $x_i \leftarrow \left\lfloor \frac{b - total\_weight}{a_i} \right\rfloor$
  $total\_weight \leftarrow total\_weight + x_i \cdot a_i$
  $total\_utility \leftarrow total\_utility + x_i \cdot c_i$
end for

Concerning the complexity of Algorithm 1, there is a first loop in $\mathcal{O}(n)$ to compute ratios, then a sorting algorithm of a list of n elements which is of worst-case complexity $\mathcal{O}(nlog(n))$. Finally we iterate through these items in the order of the sorted list of size n. This is of worst case complexity $\mathcal{O}(n)$. When combining those three parts, we obtain a final worst-case complexity for the whole algorithm of $\mathcal{O}(nlog(n))$

2.2 Dynamic programming algorithm

After some research that lead us to this article[1], we implemented the following algorithm. For the sake of simplicity, we did not add the part to save the bag composition in the pseudocode.

[1] https://www.sanfoundry.com/dynamic-programming-solutions-integer-knapsack-problem/

The dynamic programming knapsack algorithm constructs the optimal bag of size b based on the previous optimal bags. For each bag size (i in the algorithm) starting at 1, it finds the best bag composition by adding each item (j in the algorithm) to the optimal bag of size $i - w_j$ (i.e. the optimal composition that can still fit item j in bag i).

Algorithm 2 Dynamic Programming Knapsack Algorithm

for $i \in \{1..b\}$ **do**
  $obj_i \leftarrow obj_{i-1}$
  for $j \in \{1..n\}$ **do**
    $obj_i = \max(obj_i, u_j + obj_{i-w_j})$
  end for
end for

The algorithm cycles through all the n items for each of the b bag sizes, hence the time complexity of this algorithm is $\mathcal{O}(n \cdot b)$. The spatial complexity is $\mathcal{O}(b)$ if we don't store the optimal bag compositions (only the b optimal values for the objective function for each bag), and $\mathcal{O}(n \cdot b)$ if we do (for each bag size i and each item j, we have to store how many times item j is in the optimal bag of size i).

3 Result Analysis

The Different algorithms solved the problem yielding the following items packed into the knapsack:

- Gurobi Optimizer : `x = [2, 1, 0, 0, 0, 0, 1, 0, 0, 0]`
- Greedy Algorithm : `x = [3, 1, 0, 0, 0, 0, 0, 0, 0, 1]`
- Dynamic Programming Algorithm : `x = [3, 1, 0, 0, 0, 0, 0, 0, 0, 1]`

All 3 solutions are feasible. Indeed if we multiply the number of times each item is packed by its weight : $\sum_{i=1}^{n} x_i^* \cdot a_i$, we obtain 55 for all 3 results. Since b has the value 55 for this instance the weight constraint (2) is respected. Of course all items are packed an integer number of times, meaning constraint (3) is also respected.

Finally all three solutions are optimal as in this problem each utility of the object was strictly equal to its weight. Therefore the maximum utility of the whole knapsack was also its maximum weight, i.e. 55. And here we see that $z^* = \sum_{i=1}^{n} x_i^* c_i = 55 = b$ meaning that for all 3 algorithm the found solution was optimal.

4 Performance Analysis

The executions times of the 3 solvers on the 4 datasets can be found in table 1. The gurobi solver is the slowest and took more than 30 minutes for both of the large datasets, so we decided to stop it. This is because the solver has an exponential worst-case time complexity, but this solver has the advantage of being easy to use and only needing the definition of the problem. However some options were disabled to make a relevant comparison with the other algorithms, we might expect run times to decrease if we didn't disable those options. We also observed that if we stop the solver it does yield its current solution which happens to be the optimal one. It just didn't stop itself not knowing that it was the optimal one.

The greedy algorithm is very fast but doesn't give the optimal solution as can be seen on the medium dataset (it would be optimal if we could take fractional objects) while the dynamic programming solution has a time complexity of $\mathcal{O}(n \cdot b)$, but is very problem-specific. In the end, Dynamic Programming is the one that finds the optimal solution in a reasonable time, at the expense of having to derive an a problem-specific algorithm.

	Small		Medium		Large 1		Large 2	
	run time $[s]$	z^*	run time $[s]$	z^*	run time $[s]$	z^*	run time $[s]$	z^*
Gurobi Solver	$4.87 \cdot 10^{-3}$	55	$2.26 \cdot 10^{-2}$	14 552	Timeout	/	Timeout	/
Greedy	$3.73 \cdot 10^{-5}$	55	$2.96 \cdot 10^{-5}$	14 550	$1.17 \cdot 10^{-3}$	108 680	$1.79 \cdot 10^{-3}$	2 194 836
Dynamic Prog.	$1.12 \cdot 10^{-4}$	55	$1.76 \cdot 10^{-2}$	14 552	1.21	108 680	5.04	2 194 836

Table 1: Execution times