AM 221: Advanced Optimization Spring 2016

Prof. Yaron Singer *Lecture 17 — March 30th*

1 Overview

In the previous lecture, we saw examples of combinatorial problems: the Maximal Matching problem and the Minimum Spanning Tree problem. For both problems, we say that a "natural" greedy algorithm was able to find an optimal solution.

Today, we will study the Knapsack problem. We will see that a simple greedy algorithm is able to compute a good approximation to the optimal solution. We will then be able to improve on this algorithm using a new tool called dynamic programming.

2 Greediness and the Knapsack Problem

Let us first define the Knapsack Problem.

Input:

- n items. Each item $i \in [n]$ has:
  - cost $c_i \in \mathbb{Z}_+$
  - value $v_i \in \mathbb{Z}_+$
- budget $B \in \mathbb{Z}_+$

Goal: Find a subset S of items whose cost $c(S) = \sum_{i \in S} c_i$ is smaller than B and whose value $v(S) = \sum_{i \in S} v_i$ is maximal.

Example. Let us consider the Knapsack instance whose items are given by the following table:

	a_1	a_2	a_3	a_4
v_i	2	1	7	2
c_i	1	3	2	2

and with budget $B = 4$. Consider the following three solutions:

$$\text{Feasible} \begin{cases} \{a_1, a_2\} \\ v(a_1, a_2) & = 3 \\ c(a_1, a_2) & = 4 \end{cases} \qquad \text{Feasible} \begin{cases} \{a_1, a_3\} \\ v(a_1, a_3) & = 9 \\ c(a_1, a_3) & = 3 \end{cases} \qquad \text{Not Feasible} \begin{cases} \{a_1, a_2, a_3\} \\ v(a_1, a_2, a_3) & = 10 \\ c(a_1, a_2, a_3) & = 6 \end{cases}$$

The last solution is not feasible because it exceeds the budget. Among the first two solutions, the second is better because its value is larger. In fact it has maximal value among all feasible solutions.

Mathematical Program for Knapsack. The Knapsack problem can be written as the following Integer program

$$\max_{\mathbf{x}} \mathbf{v}^\intercal \mathbf{x}$$
$$s.t. \sum_{i=1}^{n} c_i x_i \leq B$$
$$x_i \in \{0, 1\}$$

As we will see in section, Knapsack is an NP-complete optimization problem and it is being too optimistic to hope to solve it in polynomial time. Instead we turn to the problem of designing an approximation algorithm for it.

Definition 1. *For a maximization problem, an algorithm is called α-approximation if for any input, the algorithm returns a feasible solution S such that:*

$$\frac{f(S)}{f(O)} \geq \alpha$$

where O is an optimal solution, and f evaluates the quality of the solution.

Fractional View of Knapsack. In problem set 4, you analyzed the Linear Programming relaxation of the Knapsack problem, and proved that an optimal solution to the relaxation could be obtained as follows:

- sort the items in decreasing order of densities. The density of element i is defined by the ration v_i/c_i (value per cost).
- add items to the solution one-by-one in this order as long as the sum of the costs does not exceed the budget. For the first item i which would violate the budget, only add a fraction x_i of it such that the budget constraint is tight. The fraction x_i is defined by:

$$x_i = \frac{B - \sum_{j<i} c_j}{c_i}$$

Approximation algorithm for Knapsack. The algorithm from problem set 4 for the fractional relaxation of Knapsack suggests the following greedy algorithm for the original Knapsack problem.

Algorithm 1 Greedy algorithm for KNAPSACK

```
S ← ∅
while c(S ∪ argmax_{i∉S} v_i/c_i) ≤ B do
    S ← S ∪ argmax_{i∉S} v_i/c_i
end while
return S
```

This algorithm by itself is not enough to provide a constant approximation to the optimal solution. Let us indeed consider the Knapsack instance with two items:

- item 1 has cost 1 and value 2
- item 2 has cost M and value M (for some $M > 2$)
- the budget is B

The greedy algorithm will select item 1 (its value per cost is better than item 2) and will then return since item 2 cannot be added without exceeding the budget. However it is clear that the optimal solution is to select item 2. The ratio between the value of the solution returned by the greedy algorithm and the optimal solution is $\frac{1}{M}$ which is arbitrarily small as M grows to infinity.

However, a simple modification of Algorithm 1 is a constant $(1/2)$ approximation algorithm as stated in the following theorem.

Theorem 2. *Let $S^* = \text{argmax}\{v(s), v(a)\}$, where S is the solution returned by Greedy and a is the element with highest density (value per cost) that is not in S. Then:*

$$v(S^*) \geq \frac{OPT}{2}.$$

where OPT denotes the value of an optimal solution.

Proof. Let OPT_{LP} be the value of an optimal solution for fractional knapsack problem as discussed above. Then $OPT \leq OPT_{LP}$ since the fractional knapsack problem is a relaxation of Knapsack. It is also clear from the construction of the fractional optimal solution that $OPT_{LP} \leq v(S) + v(a)$ since only a fraction $x_a \leq 1$ of the last element is added. Putting everything together, we obtain:

$$OPT \leq OPT_{LP} \leq v(S) + v(a) \leq 2 \cdot \max\{v(s), v(a)\}$$ □

3 Dynamic Programming Algorithm for Knapsack

Let us define:

- $S_{i,p}$: subset of $\{a_1, \ldots, a_i\}$ with minimal cost which has value **exactly** p.
- $c(S_{i,p}) : \begin{cases} \infty & \text{if } S_{i,p} \text{ does not exist} \\ \sum_{a_i \in S_{i,p}} c(a_i) & \text{otherwise} \end{cases}$

and let us consider the following recurrence:

- $S_{1,p} = \begin{cases} \{a_1\} & \text{if } p = v_1 \\ \_ & \text{otherwise} \end{cases}$
- $S_{i+1,p} = \begin{cases} \text{argmin}\left[c(S_{i,p}), c(a_{i+1} \cup S_{i,p-v_{i+1}})\right] & \text{if } v_{i+1} \leq p \text{ and } S_{i,p-v_{i+1}} \neq \_ \\ S_{i,p} & \text{otherwise} \end{cases}$

Algorithm 2 Dynamic programming algorithm for KNAPSACK

$S_{1,p} = \begin{cases} a_1 & \text{if } p = v_i \\ \_ & \text{otherwise} \end{cases}$
for $i \in \{1, \ldots, n\}$ **do**
    for $p \in \{1, \ldots, n \cdot V^*\}$ **do**
        $S_{i+1,p} = \begin{cases} \operatorname{argmin}\left\{c(S_{i,p}), c(a_{i+1} \cup S_{i,p-v_{i+1}})\right\} & \text{if } v_{i+1} \leq p \text{ and } S_{i,p-v_{i+1}} \neq \_ \\ S_{i,p} & \text{otherwise} \end{cases}$
    end for
end for
return $\operatorname{argmax}_{p:c(S_{n,p}) \leq B} v(S_{n,p})$

It is clear that this recurrence computes $S_{i,p}$ for all i and p. Furthermore, we only need to consider values of p in the range $\{0, \ldots, n \cdot V^*\}$ where $V^* = \max_{1 \leq i \leq n} v_i$. The optimal solution to Knapsack is then given by $\operatorname{argmax}_{p:c(S_{n,p}) \leq B} v(S_{n,p})$. This motivates the following dynamic programming algorithm for Knapsack.

It follows from the discussion above that Algorithm 2 computes an optimal solution to the Knapsack problem. The running time of this algorithm is $O(n^2 V^*)$. Even though it appears to be a polynomial-time algorithm, it is in fact only *pseudo-polynomial*: indeed, as you will see in section, the running time complexity is measured as a function of the size (number of bits) of the input. The value V^* can be written using $\log V^*$ bits, so a polynomial-time algorithm should have a complexity $O(\log^c V^*)$ for some $c \geq 1$. The complexity $O(n^2 V^*)$ is in fact exponential in the size of the input.

4 Polynomial-time Approximation Scheme for Knapsack

Even though the dynamic programming algorithm from Section 3 is not a polynomial time algorithm, it is possible to transform it into a polynomial-time approximation algorithm with arbitrarily good approximation ratio.

Algorithm 3 Polynomial-time approximation scheme for Knapsack

Require: parameter ε
$k \leftarrow \frac{\varepsilon V^*}{n}$
for $i \in \{1, \ldots, n\}$ **do**
    $v'_i \leftarrow \lfloor \frac{v_i}{k} \rfloor$
end for
return the solution given by Algorithm 2 for input $\left(\left\{(c_1, v'_1), (c_2, v'_2), \ldots, (c_n, v'_n)\right\}, B\right)$

Theorem 3. *Algorithm 3 runs in time $O(\frac{n^3}{\varepsilon})$, and returns a solution S such that:*

$$v(S) \geq (1 - \varepsilon) OPT$$

Proof. We will first prove the running time and then the approximation ratio.

Running time. As discussed above, the running time of the dynamic programming algorithm is $O(n^2 \cdot \alpha)$ where α is largest value in input, in our case:

$$O(n^2 \cdot \alpha) = O\left(n^2 \cdot \left\lfloor \frac{V^*}{k} \right\rfloor\right) = O\left(n^2 \cdot \frac{n}{\varepsilon}\right) = O\left(\frac{n^3}{\varepsilon}\right)$$

Approximation Ratio.

$$\begin{aligned} v(S) \geq k \cdot v'(S) &\geq k \cdot v'(O) \geq v(O) - k \cdot |O| \\ &\geq v(O) - k \cdot n \\ &= OPT - \varepsilon \cdot V^* \\ &\geq (1 - \varepsilon) OPT \end{aligned}$$

□

Remark. An algorithm like Algorithm 3 is called an approximation scheme: the algorithm is parametrized by ε and for any ε returns a solution which is a $(1 - \varepsilon)$ approximation to the optimal solution. It is then interesting to look at how the complexity depends on ε: it should grow to infinity has ε converges to zero since we don't know a polynomial time algorithm for Knapsack (more about that in section). When the complexity is polynomial in n and $\frac{1}{\varepsilon}$ as in Theorem 3, the scheme is called a *fully polynomial-time approximation scheme* (FPTAS).