CSE 632: Analysis of Algorithms II **Fall 2017**

Lecture 5 (09/10/2017): Knapsack Covering and Packing

Lecturer: Shi Li *Scribe: Luting Chen*

5.1 The Knapsack Packing Problem

In the knapsack problem (also named as max-knapsack problem), we have a knapsack with a limited space and are provided with a set of items, each with a size and some value. So how can we choose a subset of the items, so that we won't exceed the capacity of the knapsack and maximize the value of the chosen items? The mathematical description is the following:

Given a set of n items $[n]$, each item $i \in [n]$ has a size s_i and value v_i and the knapsack has capacity B. The goal is to find a subset $I \subseteq [N]$ s.t. $\sum_{i \in I} s_i \leq B$ and maximaize $\sum_{i \in I} v_i$.

A intuitive thinking exploits the greedy algorithm: sort items in decreasing order by unit value $\frac{v_i}{s_i}$ and keep picking items with highest unit value until the knapsack can no longer fit in more. This seemingly correct algorithm only works when we can pick a fraction of one item, however it doesn't work for integral knapsack packing problem, which is the configuration of our problem. It's easy to come up with a counter case: B = 100 and there are only two items with $s_1 = 1, v_1 = 1.001$ and $s_2 = 100, v_1 = 100$. Running the greedy algorithm, we will end up only choosing item1 with value 1.001 since item1 has a higher unit value than item2, however the optimal solution is choosing item2, which gives us value 100.

One correct way to solve this problem is using dynamic programming(DP). Define function $f\left(i, B^{'}\right) = max_{I \subseteq [n]: S(I) \leq B'} V(I)$ with $S(I) = \sum_{i \in I} s_i$ and $V(I) = \sum_{i \in I} v_i$. $f\left(i, B^{'}\right)$ represent the maximum value from the first i items that can fit into a knapsack with size $B^{'}$. As a result, $f(n, B)$ gives the answer to our problem. We can use a $n \times B$ table and recursion to solve the problem:

$$Initialize\left(i, B^{'}\right) = 0, \forall B^{'} = 0, 1, 2, \cdots, B$$

$$f\left(i, B^{'}\right) = max \begin{cases} max\left\{f\left(i-1, B^{'}\right), f\left(i-1, B^{'} - s_i\right) + v_i\right\} & \text{if } s_i \leq B^{'} \\ f\left(i-1, B^{'}\right) & \text{if } s_i > B^{'} \end{cases}$$

Here, $f\left(i-1, B^{'}\right)$ means we don't choose the ith item, while $f\left(i-1, B^{'} - s_i\right) + v_i$ means we choose the ith item. The running time for the DP algorithm is $O(nB)$ in theory for we need to compute each cell in the table. In practice, we can speed up the process by using a cache to store partial results or the bottom-up design to compute cells that are useful.

Dual Problem

The dual problem of the previous knapsack problem is compute $g(i, V)$, which means the minimized size of chosen items with value at least V. Using DP, we have the following:

$$g(0, 0) = 0$$

$$g(0, v) = \infty$$

$$g(0, v) = \infty$$

$$g(i, v) = min\left(g(i-1, v), s_i + g(i-1, v-v_i)\right)$$

After computing the whole table g, we can find the answer to the original problem by getting $max\{v : g(n, v) \leq B\}$.

5.2 Pseudopolynomial Time Algorithm

Definition 5.1 *A numeric algorithm runs in pseudopolynomial time if its running time is polynomial in the numeric value of the input, but is exponential in the length of the input.*

Since input numbers are encoded in binary, the size of input B is actually $logB$, so the time complexity $O(nB)$ is exponential in the size of the input number B, not polynomial. However if we were to assume inputs are given in unary, then $O(nB)$ would be polynomial in the size of the input.

5.3 Approximation Scheme for Knapsack Problem

The main idea of designing a polynomial DP algorithm for the knapsack problem is rounding items' values to smaller scales and the rounding error is not that great at the same time. Note that we can only round and change the value of items not their sizes because we may change the feasibility of the problem if we change sizes.

We scale v_i and only keep the integer part of the scaled value. For each new instance of scaled value $v_i^{'} = \left\lfloor \frac{v_i}{\mu} \right\rfloor$, where μ is the scale factor and is defined as $\mu = \frac{\varepsilon M}{n}$. ε is the approximation ratio and M is all items' maximum value $M = max_{i \in [n]} v_i$. For each item $v_i^{'}$, we can easily have

$$v_i^{'} \leq \left\lfloor \frac{M}{\mu} \right\rfloor = \left\lfloor \frac{n}{\varepsilon} \right\rfloor,$$

and this shows a running time complexity of $O(n \cdot n \cdot \left(\frac{n}{\varepsilon}\right)) = O(\frac{n^3}{\varepsilon})$.

Next, let's prove the algorithm is (1-ε)-approximate of the optimal solution. We define the following four terms first:
OPT: optimal solution of the original instances(item values are not scaled)
opt: the value configuration of the optimal solution with original instances.
$OPT^{'}$: optimal solution of the new instances(item values are scaled)
$opt^{'}$: the value configuration of the optimal solution with new instances.

We use the optimal solution of the new instance for our algorithm:

$$\sum_{i \in OPT^{'}} v_i \geq \sum_{i \in OPT^{'}} (v_i^{'} \mu) = \mu \sum_{i \in OPT^{'}} v_i^{'} = \mu OPT^{'}$$

$$\geq \mu \sum_{i \in OPT} v_i^{'} \geq \sum_{i \in OPT} (v_i - \mu) = \sum_{i \in OPT} v_i - |OPT|\,\mu$$

$$\geq opt - n\mu = opt - \varepsilon M \geq opt(1 - \varepsilon)$$

$|OPT|$ represent the number of items selected in the optimal solution and $|OPT| \leq n$. And another observation is $opt \geq M$ and that's why $opt - \varepsilon M \geq opt - \varepsilon opt = opt(1 - \varepsilon)$. We could make ε as small as possible to get closer to the optimal solution of the original instances, but the time complexity is $O(\frac{n^3}{\varepsilon})$, smaller ε leads to longer running time. This is a trade-off between time efficiency and result quality.

Dual Problem

Here we can no longer round the value because it may change the feasibility of the problem. We round sizes instead and carry out the DP algorithm.

5.4 PTAS and FPTAS

Definition 5.2 *PTAS(Polynomial-Time Approximation Scheme) is an algorithm which takes an instance of an optimization problem and a parameter $\varepsilon > 0$ and, in polynomial time, produces a solution that is within a factor $1 + \varepsilon$ of being optimal.*

Definition 5.3 *FPTAS(Fully Polynomial-Time Approximation Scheme) is an algorithm which takes an instance of an optimization problem and a parameter $\varepsilon > 0$ and, in FULLY polynomial time in n (the size of the problem) and in $1/\varepsilon$, produces a solution that is within a factor $1 + \varepsilon$ of being optimal.*

5.5 Weakly NP-hard and Strong NP-hard

Definition 5.4 *A problem is Weakly NP-hard when it is NP-hard only if input integers are not polynomial-bounded.*

Definition 5.5 *A problem is Strong NP-hard when it is NP-hard even if input integers are polynomial-bounded.*

If a problem is strong NP-hard, it cannot admit FPTAS. The knapsack covering problem is weakly NP-hard.