ORIE 6300 Mathematical Programming I October 29, 2014

Recitation 8

Lecturer: Chaoxu Tong *Topic: The Knapsack Problem*

The Knapsack Problem [1]

In the knapsack problem, we are given a set of n items $I = \{1, ..., n\}$, where item i has value v_i and size s_i, and a knapsack of capacity B. We assume all values and sizes are positive integers. We want to find a subset of items $S \subseteq I$ to put in the knapsack that maximizes the value $\sum_{i \in S} v_i$ subject to the capacity constraint of the knapsack, i.e. $\sum_{i \in S} s_i \leq B$. The decision version of the Knapsack Problem is NP-complete.

Dynamic Program for the Knapsack Problem

We can solve the knapsack problem using dynamic programming. We maintain an array entry $A(j)$ for $j = 1, \ldots, n$. Each entry $A(j)$ is a list of pairs (t, w). A pair (t, w) in the list $A(j)$ indicate that there is a subset S of the first j items such that $\sum_{i \in S} s_i = t$ where $t \leq B$, and $\sum_{i \in S} v_i = w$. However, each list does not need to contain all possible such pairs and only keep track of the most efficient ones. More formally, we say a pair (t, w) *dominates* the pair (t', w') if $t \leq t'$ but $w \geq w'$, i.e. the solution (t, w) takes less space but has more value than (t', w'). In each list $A(j)$, we only keep track of *non-dominated* pairs. Hence of $A(j)$ has entries $(t_1, w_1), \ldots, (t_k, w_k)$ with $t_1 < \cdots < t_k$ and $w_1 < \cdots < w_k$. Now we can describe the dynamic program:

1. Start with $A(1) = \{(0, 0), (s_1, w_1)\}$
2. For each $j = 2, ..., n$:
   first set $A(j) \leftarrow A(j-1)$
   for each $(t, w) \in A(j-1)$, if $t + s_j \leq B$ then add $(t + s_j, w + v_j)$ to $A(j)$
   remove dominated pairs from $A(j)$
3. return the pair (t, w) from $A(n)$ with the maximum value

We will skip the proof of correctness of this algorithm, which can be done using induction on j. Now for the running time: Computing each entry of the dynamic program takes $O(1)$ time, so it suffices to bound the size of the size of the dynamic program table. Since all inputs are integers, we can bound the size of each list $A(j)$ in the table by the number of distinct values (t, w) can take in each coordinate. Notice t can possibly take on values from 0 to B. Now let $V = \sum_{i=1}^{n} v_i$ then V gives an upper bound for value w can take, i.e. w can possibly take on values from 0 to V. Hence the number of entries in the table, and thus the running time of the algorithm is $O(n \min(B, V))$. In general, this is not polynomial to the input size (for example, the input B is encoded in binary and has size $\log_2 B$), which is expected as the knapsack problem is NP-hard. However, note that if V is polynomial in n then the running time would indeed be a polynomial in input size.

[1] Based on previous note by Maurice Cheung

FPTAS for the Knapsack Problem

The key idea is that we can round the values of the items so that V is polynomial in n. While rounding induces some loss of precision in the value of the solution, we will show this does not affect the optimal value by too much. For any given instance, let OPT denote its optimal value.

Definition 1 *A* polynomial-time approximation scheme*(PTAS) is a family of algorithms $\{A_\epsilon\}$, where there is an algorithm for each $\epsilon > 0$, such that $\{A_\epsilon\}$ is a $(1+\epsilon)$-approximation algorithm (for minimization problems) or a $(1-\epsilon)$-approximation algorithm (for maximization problems).*

That means the solution S returned by the algorithm $\{A_\epsilon\}$ has value at most $(1+\epsilon)OPT$ for minimization problems, and at least $(1-\epsilon)OPT$ for maximization problems. In general, the running time of $\{A_\epsilon\}$ is allowed to depend arbitrarily on $1/\epsilon$. If the running time of $\{A_\epsilon\}$ is bounded by a polynomial in $1/\epsilon$, we call it a *fully* polynomial-time approximation scheme (FPTAS). In some sense a FPTAS is the best one can do for a NP-hard problem since given an error parameter ϵ, one can find an approximate solution in polynomial time. However, in practice the running time of many FPTAS is very slow even for reasonable values of ϵ.

We can now describe a FPTAS for Knapsack. Let μ be an appropriate scaling factor to be defined later. Set $v'_i = \lfloor v_i/\mu \rfloor$. Then we run the dynamic programming algorithm in the previous section of the instance with item size s_i and values v'_i.

Claim 1 *Let S be the set of chosen items in the optimal solution of the rounded instance using scaling factor μ and O be the set of chosen items in an optimal solution of the original instance. There exists a scaling factor μ such that $\sum_{i\in S} v_i \geq (1-\epsilon)\sum_{i\in O} v_i$.*

Proof: Let $\hat{v_i} = \mu\lfloor \frac{v_i}{\mu} \rfloor$. Note that $0 \leq v_i - \hat{v_i} < \mu$. Hence:

$$\begin{aligned}
\sum_{i\in S} v_i &\geq \sum_{i\in S} \hat{v_i} \\
&\geq \sum_{i\in O} \hat{v_i}, \text{ (by optimality of } S \text{ in rounded instance)} \\
&> \sum_{i\in O} v_i - |O|\mu \\
&\geq \sum_{i\in O} v_i - n\mu.
\end{aligned}$$

Hence we need $n\mu \leq \epsilon OPT$. Let $M = \max_i v_i$, then M is a (trivial) lower bound on OPT. Hence setting $\mu = \epsilon M/n$ give us the desired result . □

Notice for any item j, $v'_j \leq v_j/\mu = nv_j/(\epsilon M) \leq n/\epsilon$. Hence $V' = \sum_{i=1}^{n} v'_i = O(n^2/\epsilon)$. Thus the running time of the algorithm is $O(n\min(B, V')) = O(n^3/\epsilon)$. Combining with the claim above, this means this algorithm is indeed a FPTAS.