CMPUT 675: Approximation Algorithms **Winter 2005**

Lecture 6: Feb 2

Lecturer: Mohammad R. Salavatipour *Scribe: Zhipeng Cai*

6.1 Knapsack

Today we will develop an FPTAS for the knapsack problem.

Knapsack Problem
Input: Set of items $\{1, ..., n\}$, each item has value v_i and a weight w_i, we have a knapsack with capacity B, $v_i, w_i, B \in Z^+$.
Goal: Find a subset S of items such that $\sum_{i \in S} v_i$ is maximum and $\sum_{i \in S} w_i \leq B$.

Clearly, if the weight of a single item is larger than B we can ignore that item. So let's assume $\forall i : w_i \leq B$.

The obvious greedy algorithm is to sort in non-increasing order of $\frac{v_i}{w_i}$ and pick the objects in this order. Unfortunately this greedy algorithm has a bad solution. In fact, knapsack is an NP-hard optimization problem. First we present a dynamic Programming algorithm for knapsack. Let V be the largest value among all items. Clearly

$$OPT \leq nV.$$

We have an $n \times (nV + 1)$ table A. Let $A[i, v]$ be the smallest weight of a subset of items from $\{1, \ldots, i\}$ such that the value of the items is exactly equal to v, if no such set exists then $A[i, v] = \infty$. Then $A[1, v]$ is easy to compute for any value $v \in \{1, \ldots, nV\}$ and

$$A[i, v] = \begin{cases} \min\{A[i-1, v], A[i-1, v - v_i] + w_i\} & \text{if } v_i < v \\ A[i-1, v] & O.W. \end{cases}$$

This leads to the following algorithm:

Dynamic Programming algorithm for Knapsack

$V = \max_{1 \leq i \leq n} V_i$
for $i = 1$ to n do $A[i, 0] = 0$
for $v = 1$ to nV do

$$A[1, v] = \begin{cases} w_1 & if \quad v_1 = v \\ \infty & \texttt{otherwise} \end{cases}$$

for $i = 2$ to n do
  for $v = 1$ to nV do
    if $v_i \leq v$ then
      $A[i, v] = \min\{A[i-1, v], A[i-1, v - v_i] + w_i\}$
    else
      $A[i, v] = A[i-1, v]$

Figure 6.1: Dynamic Programming Algorithm for Knapsack.

The running time of this algorithm is $O(n^2 \cdot V)$. But we know that knapsack is NP-hard. So have we proved that P=NP?! This is NOT polynomial in the size of input because V is not polynomial in size of v_i's (represented in binary). We only need $\log V$ bits to represent V, so n^2V is exponential in V. This is polynomial only if the input is given in unary representation. For this reason, we call this algorithm a *pseudo-polynomial* time algorithm.

The main reason that Dynamic Programming is not polynomial time is that the values of items can be much larger than n. If they were all polynomially bounded by n, then this algorithm would be polynomial time. We are going to use this fact to design an FPTAS for knapsack. To do so, we are going to use only a polynomially bounded segments of values that will depend on n and $\frac{1}{\varepsilon}$ (the error parameter). Then we will find a solution (in polynomial time) that is at least $(1-\varepsilon).OPT$ using dynamic programming.

FPTAS for knapsack:

- Let $k = \frac{\varepsilon V}{n}$.
- For each i, let $v_i' = \lfloor \frac{v_i}{k} \rfloor$.
- Run Dynamic Programming with (w_i, v_i').
- Let S' be the solution, return S'.

Theorem 6.1 *This is an FPTAS for knapsack.*

Proof: Let S be an optimal solution and OPT be the value of this solution. For each item i: $kv_i' \leq v_i \leq k(v_i'+1)$. Therefore:

$$v_i - k \leq kv_i' \tag{6.1}$$

and

$$v_i \geq kv_i' \tag{6.2}$$

So by (6.1): $OPT = \sum_{i\in S} v_i \leq k.\sum_{i\in S} v_i' + kn$.

Since S' is an optimal solution for values v_i''s, therefore, for any set, and in particular for S:

$$\sum_{i\in S'} v_i' \geq \sum_{i\in S} v_i' \tag{6.3}$$

Thus:

$$\begin{array}{lll} \sum_{i\in S} v_i & \geq k.\sum_{i\in S'} v_i' & \text{by (6.2)} \\ & \geq k.\sum_{i\in S} v_i' & \text{by (6.3)} \\ & \geq \sum_{i\in S} v_i - k.n & \text{by (6.1)} \\ & = OPT - kn & \\ & = OPT - \varepsilon V \geq (1-\varepsilon)Opt & \end{array}$$

because εV is an upper bound for OPT. This shows that the value of the solution found is at least $(1-\varepsilon)Opt$ The running time of this algorithm is $O(n^2 \lfloor \frac{V}{k} \rfloor) = O(n^2 \lfloor \frac{n}{\varepsilon} \rfloor = O(\frac{n^2}{\varepsilon})$. Therefore, this algorithm is an FPTAS. ■

Definition 6.2 *A problem is strongly-NP-hard if there is a reduction from every problem in NP to that problem such that every number in the reduction is in unary representation.*

It follows from the following theorem that strongly-NP-hard problems do not have pseudo-polynomial algorithms. Most NP-hard problems are in fact strongly-NP-hard. There are a few exceptions, like knapsack.

Let $|I_u|$ denote the size of an instance I in unary representation.

Theorem 6.3 *Suppose* Π *is an NP-hard minimization problem such that the objective function is integer on any instance* I *and* $Opt(I) \leq P(|I_u|)$ *where* $P(\cdot)$ *is some polynomial. Then if* Π *has an FPTAS, it is not strongly-NP-hard (i.e., it has a pseudo-polynomial algorithm).*

Proof: Suppose that Π has an FPTAS, called A. Let $\varepsilon = 1/P(|I_u|)$ and assume that the running time of A is $Q(n, 1/\varepsilon)$ where Q is some polynomial. Run A with this ε. The value of solution returned is at most $(1+\varepsilon)OPT = OPT + \varepsilon OPT < OPT + \varepsilon P(|I|_u) = OPT + 1$. Because the solution is an integer, therefore the solution by A is indeed the optimal and is found in time $Q(n, P(|I_u|))$, which is polynomial. Therefore A with this parameter ε is a pseudo-polynomial time algorithm. ■

6.2 Bin Packing

Bin packing problem

Input: Set $S = \{1, ..., n\}$ of items, each with size $s_i \in Q^+$.
Goal: Find a minimum number of unit size bins into which these items can be packed, i.e., partition items into k groups $G_1, G_2, ..., G_k$, such that $\forall 1 \leq j \leq k : \sum_{i \in G^k} s_i \leq 1$ and minimize k.

Theorem 6.4 *There is no* α*-approximation algorithm for Bin Packing with* $a < 3/2$ *unless P=NP.*

Proof: Consider the following NP-hard problem.

Partition:

Input: set of items $S = \{1, ..., n\}$ with size $(0 \leq s_i \leq 1) \in Q^+$.

Question: Can we partition S into two parts S' and $S - S'$ such that $\sum_{i \in S} S_i = \sum_{j \in S - S'} S_j$?

Let I be an instance of partition. Scale all S_i's such that $\sum S_i = 2$ and let this instance I' be the input to Bin Packing. If all items of I' fit into 2 bins, since their total sum is 2, both bins must be full and therefore I is a yes instance (the partition is given by the items in 2 bins for I'). On the other hand, if I is a Yes instance, then the corresponding partition implies that the set of items in each part can be fit into one bin for the corresponding instance I'. Therefore, the set of items of I' can be fit into 2 bins if and only if I is a Yes instance. So if we can distinguish between 2 and ≥ 3 for I' then we can decide between Yes and No for I. Therefore, there is no better than $\frac{3}{2}$-approximation for bin packing unless P=NP. ■