Approximation Algorithms - Tools and Techniques **Spring 2002**

Lecture 5 : Jan 25, 2002

Lecturer: Kamesh Munagala [1] *Scribe: Siddartha Y R*[2]

0.1 The KNAPSACK Problem

Given a set of objects $\{O_1, O_2, \ldots, O_n\}$, whose corresponding sizes and profits(value) are, respectively $\{s_1, s_2, \ldots, s_n\}$ and $\{p_1, p_2, \ldots, p_n\}$ and a knapsack of size B, to find a maximal (w.r.t. profit) set of objects whose total size is not greater than that of the knapsack. i.e, a set S of objects which

$$\max_S \sum_{i \in S} p_i$$

$$\text{s.t.} \sum_{i \in S} s_i \leq B.$$

A greedy algorithm would order the objects in decreasing order of thier relative profits given by p_i/s_i, and pick objects greedily w.r.t. thier relative profits. But, this algorithm can be made to perform arbitrarily badly, as shown by the problem instance shown in Figure (1). By assigning a *slightly* larger relative profit to O_1 , the greedy algorithm is forced to pick O_1, though the optimum solution is to pick O_2 since it's absolute profit (because of it's size) is larger. By making the size of O_1 very small, the solution returned by the greedy agorithm can be made arbitrarily bad.

Definition 1 *A* Polynomial Time Approximation Scheme *(PTAS) is an algorithm which, given a problem instance and any* constant ϵ*, finds a solution within* $(1 \pm \epsilon)$ *of the optima (* $\pm$ *for maximisation/minimisation problems respectively) in time polynomial in the size of input.*

Definition 2 *A* Fully Polynomial time Approximation Scheme *(FPTAS) is an algorithm which, given a problem instance and* any ϵ*, returns a solution within* $(1 \pm \epsilon)$ *of the optima (* $\pm$ *for maximisation/minimisation problems respectively) in time polynomial in the size of input.*

[1]Dept of Comp. Sci., Stanford Univ.

[2]Dept of Comp. Sci. and Automation, Indian Institute of Science, Bangalore.
`siddu@csa.iisc.ernet.in`

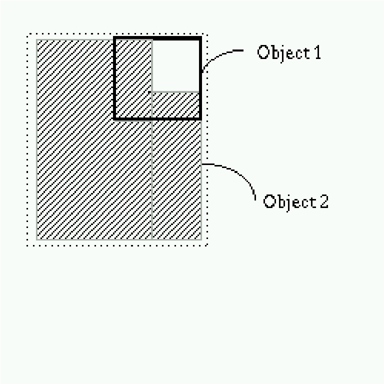

Figure 1: A bad instance for the Greedy Algorithm

Note that for a FPTAS any fixed ϵ can be specified in the input, not necessarily a constant. In particular, it may be a (polynomial time) computable function of the size of input. For many problems, it can be shown that a PTAS/FPTAS do not exist (unless $\mathcal{P} = \mathcal{NP}$). Eg. PTAS for TSP and FPTAS for MAX-CUT.

Exercise 1 *Show that there exists no FPTAS for MAX-CUT unless $\mathcal{P} = \mathcal{NP}$.*

Dynamic Programming
Solve a problem in a recursive fashion by finding solutions to subproblems and combining them. The subproblems are typically created by heavily restricting a "resource" so that the feasible space is very small and hence solution is trivial. Then, the restriction in relaxed in stages and the subproblems so created are solved *using the solutions computed* for the previous stages. These solutions to subproblems are maintained in a table to avoid repeatedly solving the same subproblem. The recursive relation between a subproblem and its sub-subproblems must be such that the combining operation is fast.

0.2 FPTAS for KNAPSACK

Here, we show an FPTAS for KNAPSACK using a dynamic programming technique. The Table of the dynamic program is shown in Figure(2). The subproblem $S(i,p)$ coresponding to cell (i,p) in the table is given by the (size of) minimal size subset of the first i objects (ordering the objects arbitrarily) which yields a profit of at least p.

$$\min_{S \in \{O_1, O_2, \ldots, O_i\}} \sum_S s_i$$

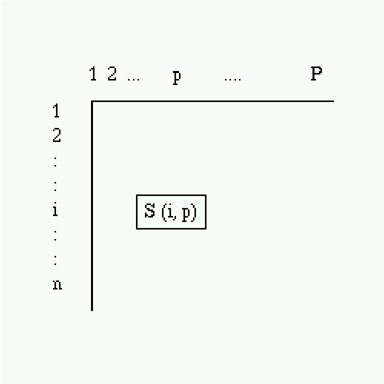

Figure 2: Table for Knapsack Dynamic Program. where $P = \sum_i p_i$

$$\text{s.t.} \sum_S p_i \geq p$$

Computing the entries in the table is straightforward. It can be done in constant time using previously computed entries in the table. Consider the subproblem of cell (i, p). The size of the minimal size subset for this subproblem depends upon whether or not it includes the object O_i. The corresponding size of the subset is then

$$S(i,p) = S(i,p) = S(i-1, p-p_i) + s_i \text{ or } S(i-1,p).$$

Hence, $S(i,p)$ can be obtained as

$$S(i,p) = \min(S(i-1,p),\ S(i-1,p-p_i) + s_i).$$

The entries of the first row and column, the "boundary conditions", as it were, are given by

$$S(1,p) = \begin{cases} 0 & ;\ p \leq 0 \\ s_1 & ;\ 0 < p \leq p_1 \\ \inf & ;\ p > p_1 \end{cases}$$

and

$$S(i,p) = 0; \quad \forall i,\ p \leq 0.$$

The Algorithm for solving KNAPSACK (call it DynKnapsack) essentially fills in entries in the above table, until the sizes in all cells of a row become greater than the knapsack size B. Then, the set of maximal profit with $S(i,p) \leq B$ is returned. The running time is dominated by the time taken to compute the entries in the table.

$$\begin{aligned} \text{Running time } &= \text{Size of table} \times \text{Time to compute a cell}(= O(1)) \\ &= n \textstyle\sum_i O(1) p_i \\ &= \ O(n \textstyle\sum_i p_i) \end{aligned}$$

Let p_{max} be the maximal profit among all the objects. Then

$$\sum_i p_i \leq np_{max}$$

and

$$\text{Running Time} = O(n^2 p_{max}).$$

Note that the running time not only depends on a measure of the size of the instance (n), but also on the actual values specified in the problem (due to the term p_{max}). Now consider the input to the algorithm. If the problem instance $\{\{s_1, s_2, \ldots, s_i\}, \{p_1, p_2, \ldots, p_i\}, B\}$ is encoded in a *unary representation* [3], then the value p_{max} is itself proportional to the size of the input, and hence, the algorithm may be regarded as having running time polynomial in size of input. However, if the problem is encoded in a *binary representation*, then, let $\langle p_{max} \rangle$ denote the representation of p_{max}, and $|\langle p_{max} \rangle|$ denote its size in this representation. Then,

$$p_{max} = O(2^{|\langle p_{max} \rangle|})$$

and the running time becomes exponential in the length of the input. This is an indication that a particular problem has a weaker form of $\mathcal{NP}$-hardness in that the hardness depends upon the input representation. In contrast, there are the problems which (provably) do not have polynomial time algorithm, even when input size is that of a unary representation. The class $\mathcal{SNP}$, for STRONGLY $\mathcal{NP}$-hard is a subset of this class of problems. Most commonly encountered $\mathcal{NP}$-hard problems are actually $\mathcal{SNP}$-hard. Eg. MAX-CUT$\in$ $\mathcal{SNP}$-hard.

Definition 3 *A* Pseudo-polynomial Time Algorithm *is one whose running time is polynomial in the size of it's input* in unary form.

What we have seen so far is an *exact* algorithm for KNAPSACK, albeit one which takes exponential time if the input is not in unary form (as inputs tend to be).

An intuitive way to overcome the above problem of running time depending on input values is to map the profits from a large values to smaller values. Next, we see what effect such a mapping has on the performance of our algorithm. Without loss of generality, we consider problems with integral profits, and define the mapping from the range of profits $\{1, 2, \ldots, p_{max}\}$ to the reduced range $\{1, 2, \ldots, k\}$, with $k < p_{max}$, defined by

$$f(p_i) = \left\lfloor \frac{p_i}{p_{max}} k \right\rfloor$$

Let the optimal solution for a problem be $OPT = \{O_{i1}, O_{i2}, \ldots, O_{im}\}$ and the solution returned by DynKnapsack be $DYN = \{O_{j_1}, O_{j_2}, \ldots, O_{j_n}\}$(In the following, subscript i indicates

[3] By unary, we mean that the base of representation is one.

an optimal solution and subscript j indicates solution returned by DynKnapsack.). Since OPT is optimal,

$$\sum_{DYN} \left\lfloor \frac{p_j}{p_{max}} k \right\rfloor \geq \sum_{OPT} \left\lfloor \frac{p_i}{p_{max}} k \right\rfloor$$

To obtain an approximation ratio, we need to find the relation between $\sum_{OPT} p_i$ and $\sum_{OPT} p_j$.

Since by definition, for any $x, y \neq 0$

$$\frac{x}{y} - 1 \leq \left\lfloor \frac{x}{y} \right\rfloor \leq \frac{x}{y}$$

By multiplying both sides of Equation. 0.2 with p_{max}/k, we get,

$$\begin{aligned} \text{LHS} \quad & \leq \textstyle\sum_{DYN} \left\lfloor \frac{p_j}{p_{max}} k \right\rfloor \frac{p_{max}}{k} \\ & \leq \textstyle\sum_{DYN} p_j \end{aligned}$$

and,

$$\begin{aligned} \text{RHS} \quad & \geq \textstyle\sum_{OPT} \left\lfloor \frac{p_i}{p_{max}} k \right\rfloor \frac{p_{max}}{k} \\ & \geq \textstyle\sum_{OPT} \left(\frac{p_{max}}{k} . \frac{p_i k}{p_{max}} - \frac{p_{max}}{k} \right) \\ & = \textstyle\sum_{OPT} p_i - \sum_{OPT} \frac{p_{max}}{k} \\ & \geq \textstyle\sum_{OPT} p_i - n \frac{p_{max}}{k}. \text{ in the worst case} \end{aligned}$$

Therefore we have,

$$\textstyle\sum_{DYN} p_j \quad \geq \sum_{OPT} p_i - \frac{n p_{max}}{k}$$

i.e.,

$$C \quad \geq C^* - \tfrac{n p_{max}}{k}$$

To obtain a FPTAS, we need to have, for a given ϵ,

$$C \geq C^*(1 - \epsilon).$$

Therefore, given an ϵ, by setting $k = \lfloor \frac{n}{\epsilon} \rfloor$, we have the required approximation bound. The running time

$$= O(n^2 k) = O(n^2 \frac{n}{\epsilon}) = O(\frac{n^3}{\epsilon}).$$

satisfies the requirement for a FPTAS.