College of Computer Science
Northeastern University
COM 3391: Advanced Algorithms

Lecture 3, part 1
13 October 1999
Scribe: Baback Elmieh

Fully Polynomial Time Approximation Schemes

1 Introduction

In our study of approximation algorithms so far we have seen several problems for which an absolute performance guarantee was not possible. We attempted to relax the absolute constraint by considering polynomial time approximation schemes (PTAS) that - particularly in the case of KNAPSACK - allowed us to get as close an approximation as we wanted by trading computation time for accuracy. Our PTAS algorithm for KNAPSACK had a time complexity of $O(\frac{n^{2+\frac{1}{\epsilon}}}{\epsilon})$.

It turns out that we can actually do even better than this for some particular problems. In the case of KNAPSACK we will see that there actually exists an algorithm that can run in $O((\frac{n}{\epsilon})^c)$ for some constant c. In other words, we can find an algorithm that is polynomial in both n and $\frac{1}{\epsilon}$.

Definition 1 (Fully Polynomial Time Approximation Scheme - FPTAS) *A* **FPTAS** *for a problem Π is an algorithm $\mathcal{A}$ that takes as input instance I of Π and an error bound ϵ and returns a solution $\mathcal{A}(I,\epsilon)$ s.t.*

$$\frac{\mathrm{OPT}(I)}{\mathcal{A}(I,\epsilon)} \leq 1+\epsilon$$

The running time of $\mathcal{A}(I,\epsilon)$ is polynomial in both $|I|$ and $\frac{1}{\epsilon}$.

Before we look at an FPTAS for KNAPSACK we will first look at an optimal algorithm to KNAPSACK which we will modify and use in our formulation of the FPTAS.

2 Optimal solution for Knapsack - First Attempt

Recall that 0/1 KNAPSACK is defined as:

- **Instance:** n items with sizes $s_1, s_2, ..., s_n$, profits $p_1, p_2, ..., p_n$ and capacity B.
- **Solution:** Select subset K of items s.t. $\sum_{i \in K} s_i \leq B$.
- **Optimize:** Maximize $\sum_{i \in K} p_i$.

The following dynamic programming algorithm solves the 0/1 KNAPSACK problem optimally:

Table size: $B * n$, the rows correspond to size, and columns to items.

$$K[i,j] = \begin{cases} \text{The Max profit set of items picked from } \{1,...,i\} \text{ with total size exactly } j. \\ 0, \text{ if no subset of } \{1,...,i\} \text{ adds to total size } j. \end{cases}$$

Boundary condition: $K[i,0] = \emptyset$ for $i = 1,...,n$.

To define the recursive function we note that there are basically two cases for setting $K[i,j]$, if

$i \in$ the set of solutions then we set $K[i,j] = K[i-1, j-s_i] \cup \{i\}$, otherwise, if $i \notin$ the set of solutions then we pick $K[i,j] = K[i-1,j]$. For an optimal solution we need to pick a set that yields the maximum profit:

$$\max(PROFIT(K[i-1,j]), PROFIT(K[i-1,j-s_i]) + p_i).$$

So for determining $K[i,j]$ in our table we need to look one cell up at $K[i-1,j]$ and one cell back and up at $K[i-1,j-1]$. To read the final OPT solution we look along the final row of our table and pick the set of items in the first non-empty cell going from right to left. The following figure illustrates the above points:

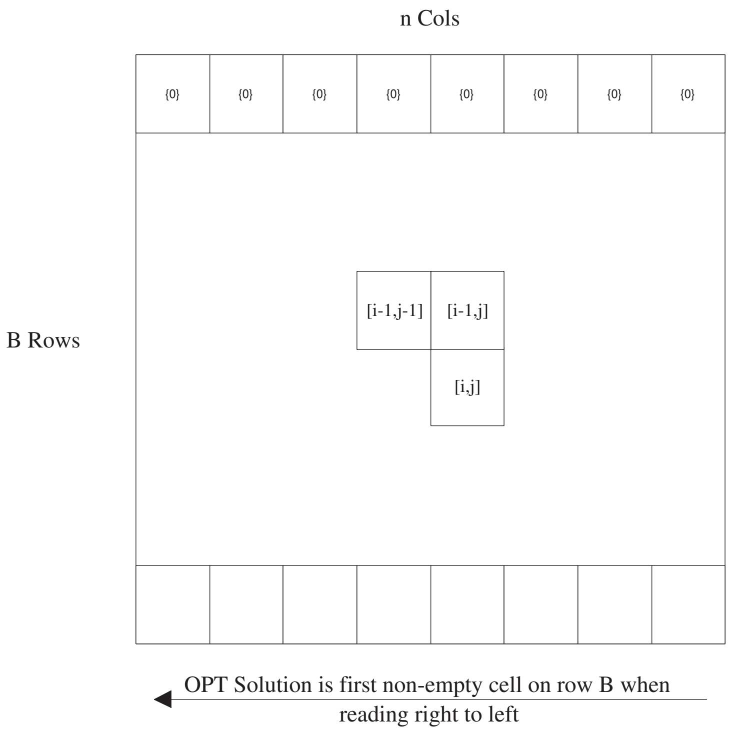

Figure 1: DP table for OPT KNAPSACK

The running time of this algorithm is $O(n^2 B)$, which is not a polynomial time algorithm since it is not polynomial in the size of B which is $\log(B)$. This form of running time complexity is known as *pseudo-polynomial.*

Definition 2 (Pseudo-polynomial algorithms) *An algorithm that solves a problem π is called a pseudo-polynomial algorithm for π if its time complexity is bounded above by a polynomial function of the two variables $|I|$ and P_{max}, where P_{max} is longest integer that appears in the instance.*

Pseudo-polynomial algorithms are the essential building blocks for FPTAS. Note that by definition any polynomial algorithm is also pseudo-polynomial because it runs in time bounded by a polynomial in $|I|$ alone. Also as we have just seen for KNAPSACK even though NP-completeness rules out the possibility of solving $\pi \in NP$ with a polynomial time algorithm (unless P = NP) it does not rule out the possibility of solving π with a pseudo-polynomial algorithm.

Unfortunately the version of optimal KNAPSACK just presented does not do so well for converting to an FPTAS, so we first need to modify it slightly in order to formulate our FPTAS for KNAPSACK.

3 Optimal Solution for Knapsack - Second Attempt

Table size: $n * (n * P_{max})$, where $P_{max} = \max_i\{p_i\}$

$$S[i,j] = \begin{cases} \text{Smallest set of items taken from } \{1,...,i\} \text{ with total profit} = j. \\ 0 \text{ if no subset of } \{1,...,i\} \text{ adds to total profit } j. \end{cases}$$

Boundary condition: $S[i,0] = \{0\}$ for $i = 1,...,n$

For our recursive function, again we note two cases for $S[i,j]$: If $i \notin$ set of solutions then $S[i,j] = S[i-1,j]$. Else, if $i \in$ set of solutions we set $S[i,j] = S[i-1,j-p_i] \cup \{i\}$. Since we want the smallest set we pick the minimum in terms of total size of these two cases:

$$\min(|S[i-1,j]|, |S[i-1,j-p_i] \cup \{i\}|).$$

This algorithm runs in $O(n^3 P_{max})$, which obviously will work well for small P_{max} but not so well if $P_{max} = 10^6$, for example.

4 FPTAS for Knapsack

Our strategy for building an FPTAS for KNAPSACK is simple: given an instance I of KNAPSACK we will transform it to I' s.t. $P_{max} \in I'$ is small. We will then solve I' optimally using the above DP algorithm and transform $OPT(I')$ to a solution for I.

FPTAS-KNAPSACK (Given instance I and k)

- Build instance I_k from I in which $\forall i\ p_i \to \frac{p_i}{k}$.
- Solve I_k optimally using DP for KNAPSACK.

If we had assumed $p_i \in \mathbf{R}$ then the set of items making up the OPT solution for I_k would have been the same set of items that made up the OPT solution for I. However, we know that $p_i \in \mathbf{Z}$ so we need to modify step 1 of FPTAS-KNAPSACK to a more reasonable scaling procedure, such as $p_i \to \lfloor \frac{p_i}{k} \rfloor$. But doing this will no longer allow us to say that $OPT(I_k) = OPT(I)$. But it turns out that this is actually not so bad as an approximation.

Analysis of FPTAS for Knapsack

Let $\forall i\ p_i \to \lfloor \frac{p_i}{k} \rfloor$ for I_k, S be the OPT solution for I_k, and S^* be the OPT solution for I.

We need to see how far S is from S^*, we can immediately claim:

$$\sum_{i \in S} \left\lfloor \frac{p_i}{k} \right\rfloor \geq \sum_{i \in S^*} \left\lfloor \frac{p_i}{k} \right\rfloor.$$

Multiplying both sides by k gives us:

$$\begin{aligned}\sum_{i\in S} p_i &\geq \sum_{i\in S^*} k\cdot\left\lfloor\frac{p_i}{k}\right\rfloor\\ &\geq \sum_{i\in S^*} k\cdot(\frac{p_i}{k}-1)\\ &\geq (\sum_{i\in S^*} p_i) - k\cdot n.\end{aligned}$$

This is the same as saying:

$$PROFIT(S) \geq PROFIT(OPT(I)) - k\cdot n.$$

We need to know under what condition

$$PROFIT(S) \geq PROFIT(\frac{OPT(I)}{1+\epsilon}).$$

We know that this is the case when $kn \leq \epsilon\cdot PROFIT(S)$, hence:

$$PROFIT(S) + k\cdot n \geq PROFIT(OPT(I)).$$

Therefore we need to set k s.t. $k \leq \epsilon\cdot P_{max}$, so we can set it:

$$k = \frac{\epsilon\cdot P_{max}}{n}.$$

This gives us an algorithm that can come within ϵ of $OPT(I)$ and has running time of

$$O(n^3\cdot P_{max}).$$

Since the maximum profit in the scaled instance is $O(\frac{n}{\epsilon})$, we have a $(1+\epsilon)$ approximation for KNAPSACK in time $O(\frac{n^4}{\epsilon})$.

In pages 364-365 of [Hoc95], a faster FPTAS for KNAPSACK due to Ibarra and Kim is presented which runs in $O(n\log_2(\frac{1}{\epsilon}+\frac{1}{\epsilon^4}))$.

■

References

[Hoc95] D. S. Hochbaum, editor. *Approximation Algorithms for NP-hard Problems.* PWS Publishing Company, Boston, MA, 1995.