College of Computer Science
Northeastern University
COM 3391: Advanced Algorithms

Lecture 2, part 1
29 September 1999
Scribe: Daniel Reinhart

Introduction to Approximation Algorithms

1 Introduction

Algorithm $\mathcal{A}$ is a polynomial time algorithm for problem Π, if the running time of $\mathcal{A}$ on an input instance I of Π, is polynomial in time to the size of I. That is, if $|I| = n$, then the running time of $\mathcal{A}$ is $O(n^c)$, for some constant $c \geq 0$. The size of I is equal to the length of its representation. The most common representation is what we refer to here as the *standard encoding.*

Definition 1 (Standard Encoding) *The following rules define the standard encoding format.*

- *Numbers are written in binary.*
- *Lists are written as a sequence of its elements separated by commas and enclosed by braces.*

■

Example 1 *Number of bits used in the standard encoding of a graph $G = (V, E)$.*

$$\begin{array}{rcllll} n &=& |V| & & & \\ m &=& |E| & & & \\ \#bits &\approx& n\lg n + 2m\lg n &=& L & \\ & & & m^5 &\approx& L^c \text{ (for some constant } c) \end{array}$$

Thus, an algorithm that runs in time $O(m^5)$ is polynomial in the size of the graph. ■

The particular encoding of the input plays a crucial role in the determining the complexity of an algorithm. This is well-illustrated by the problem of primality.

Definition 2 *The primality problem is defined as follows.*

- **Instance:** *A number n.*
- **Question:** *Is n prime?* ■

A simple algorithm for the problem is to check if n is divisible by any of $2, 3, \ldots, \lfloor\sqrt{n}\rfloor$. If *yes* then return *no* else return *yes.*

If we look at the encoding of an instance, we discover that the algorithm isn't polynomial time.

$$\sqrt{n} = n^{\frac{1}{2}} = 2^{\frac{\lg n}{2}} = 2^{\frac{|I|}{2}}$$

2 Knapsack

Definition 3 (Knapsack) *Knapsack is a maximization problem and is defined as follows.*

- **Instance:** *Given* n *items with sizes* $s_1, s_2, \ldots, s_n$*, profits* $p_1, p_2, \ldots, p_n$*, and sack capacity of* B*.*
- **Solution:** *Select subset* K *of the items such that* $\sum_{i \in K} s_i \leq B$*.*
- **Optimization:** *Maximize* $\sum_{i \in K} p_i$*.* ■

Knapsack is an NP-complete problem. A proof can be found in [**?**]. Given this result, it is unlikely that the knapsack problem can be solved optimally in polynomial time. Can we provide an *absolute performance guarantee*; that is, can we design an algorithm that gets us within k of the optimal solution, for some fixed value of k?

Definition 4 (Absolute Performance Guarantee) *An approximation algorithm* $\mathcal{A}$ *for* Π *has an absolute performance guarantee of* k*, if for all instances* I *of the problem* Π*, we have*

$$|\mathcal{A}(I) - OPT(I)| \leq k.$$

For a maximization problem, the above inequality is equivalent to $\mathcal{A}(I) \geq OPT(I) - k$*, while for a minimization problem, it is equivalent to* $\mathcal{A}(I) \leq OPT(I) + k$*.* ■

Theorem 1 *If Knapsack has an absolute performance guarantee for any fixed* k*, then Knapsack can be solved optimally in polynomial time.*

Proof: Given algorithm $\mathcal{A}'$ that polynomial time solves Knapsack within k of the optimal, we will now construct an algorithm $\mathcal{A}$ that solves the problem optimally. Algorithm $\mathcal{A}$ takes instance I and transforms it into instance I', then returns the solution returned by $\mathcal{A}'(I')$.

The transformation of I to I' is by means of *scaling* the profits. The instance I' consists of the same set of elements as I, each of which has the same size as that of I. We set the profit of item i in I' to $(k+1)p_i$. (Scaling is a useful technique in relating instances of a problem; it will appear many times during this course.)

Since I and I' differ only in the profits which are scaled by $k+1$ times for I', it follows that $\mathcal{A}(I)$ differs from $\mathcal{A}'(I')$ only by $k+1$, the factor used to blow up the profits. The optimal solution for I differs from the optimal for I' by the same factor. We thus have

$$\begin{aligned} \mathcal{A}(I) &= \frac{\mathcal{A}'(I')}{k+1}, \text{ and} \\ OPT(I) &= \frac{OPT(I')}{k+1}. \end{aligned}$$

Furthermore, by our assumption, the solution returned by $\mathcal{A}'$ is as good as the optimal minus k. Therefore, we have

$$\mathcal{A}'(I') \geq OPT(I') - k.$$

The above three inequalities together imply

$$\mathcal{A}(I) \geq OPT(I) - \frac{k}{k+1}$$

So, $\mathcal{A}$ can only differ from the optimal solution by $\frac{k}{k+1}$. However, $\frac{k}{k+1} < 1$ and all solutions are integers, so $\mathcal{A}(I)$ must be equal to $OPT(I)$. ■

We will now look at various approximation algorithms for Knapsack that run in polynomial time.

Definition 5 (Greedy Algorithm (GA)) *Our first try uses the profit density of the items.*

1. *Sort items according to profit-density ($\frac{p_i}{s_i}$) in non-increasing order.*
2. *Greedily add as many item as you can skipping items that won't fit.* ■

The sort will have a running time of $O(n \lg n)$ and the addition of items $O(n)$ for a total running time of $O(n \lg n + n) = O(n \lg n)$. What is the performance guarantee of the greedy algorithm? Consider the following instance.

Item	**Size**	**Profit**
1	1	2
2	100	100
$B = 100$		

Our greedy algorithm would pick item 1 as it has the highest profit density, but we can not fit the second item into the bag. The optimal solution is to pick item 2. We thus have a ratio of 50 between the $GA's$ total profit compared to the optimal profit. It is easy to see that we can modify the above example to make this ratio as large as possible. But, is the greedy algorithm that bad?

We are now going to show $GA(I)$ is not much less than $OPT(I)$. Let $\mathcal{X}$ and $\mathcal{X}^*$ be the solutions generated by GA and the optimal solution, respectively. Let us number all the items in nonincreasing order of their profit densities. Let i denote the first item that GA could not accomodate in X. It follows that the total profit of $\mathcal{X}$ is at least $\sum_{j<i} p_j$ and the total profit of $\mathcal{X}^*$ is no more than $\sum_{j \leq i} p_j$ We thus have

$$\begin{aligned} PROFIT(\mathcal{X}) + p_j &\geq PROFIT(\mathcal{X}^*), \text{ which implies that} \\ \text{either } PROFIT(\mathcal{X}) &\geq \frac{PROFIT(\mathcal{X}^*)}{2} \\ \text{or } p_j &\geq \frac{PROFIT(\mathcal{X}^*)}{2}. \end{aligned}$$

The above analysis of the greedy algorithm suggests the following modification, which helps eliminate the bad worst-case behavior.

Definition 6 (Modified Greedy Algorithm (MGA)) *We perform the following two operations.*

1. *Run the greedy algorithm.*
2. *If the maximum profit item is greater than the profit of the greedy algorithm return the maximum profit item otherwise return the greedy algorithm solution.* ■

Theorem 2 *The modified greedy algorithm has an approximation ratio of at most 2.* ■

Definition 7 (Approximation Ratio) *The approximation ratio of algorithm $\mathcal{A}$.*

$$\textit{Minimization:} \max_I \frac{\mathcal{A}(I)}{OPT(I)},$$
$$\textit{Maximization:} \max_I \frac{OPT(I)}{\mathcal{A}(I)}.$$

A guarantee based on an approximation ratio is referred to as a *relative performance guarantee.* ■

3 Multiprocessor Scheduling

Definition 8 (Multiprocessor Scheduling (MPS)) *Multiprocessor scheduling is a common type of minimization problem.*

- **Instance:** *Given m machines and n jobs with processing time $p_1, p_2, \ldots, p_n$.*
- **Solution:** *Partition n jobs into m sets.*
- **Optimization:** *Minimize the time taken to complete all jobs (makespan).*

Example 2 *Given the following instance a sample schedule is shown.*

$$\begin{aligned} m &= 2 \\ \text{job sizes} &= 1, 3, 5, 2 \\ \text{schedule} &= \{\text{jobs of size 1 and 5}\}, \{\text{jobs of size 2 and 3}\} \end{aligned}$$

The makespane of this schedule is $1 + 5 = 6$.

A simple greedy algorithm is to go through the jobs in any order and place the job on the least loaded machine. We now analyze this algorithm, which is illustrated in Figure 1.

Let I denote the given instance, and let $GA(I)$ denote the schedule produced by the greedy algorithm. Let $OPT(I)$ denote the schedule produced by the optimal solution. For convenience, we will also use $GA(I)$ and $OPT(I)$ to denote the makespans of the two schedules. Consider the machine that has maximum load in $GA(I)$. Let p_j be the last job added to this machine. The load of the machine that p_j was added to, had the least load of all machines, at the time p_j was added. By using an *averaging argument* we can find the lower bound on the optimal solution and an upper bound on the approximation algorithm.

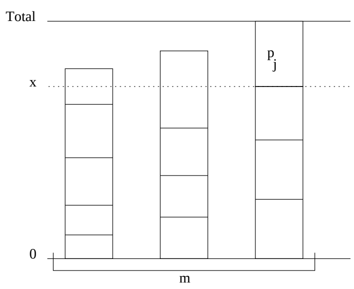

Figure 1: Addition of p_j

In Figure 1, since $x * m \leq \sum_{i \neq j} p_i$, we can place the following upper bound on $GA(I)$, the makespan of the

$$GA(I) \leq p_j + \frac{\sum_{i \neq j} p_i}{m}.$$

Furthermore, we can place the following lower bound on $OPT(I)$ using averaging.

$$OPT(I) \quad \geq \quad \frac{\sum p_i}{m}.$$

We thus have:

$$\begin{aligned} GA(I) \quad &\leq \quad p_j + \frac{\sum p_i}{m} - \frac{p_j}{m} \\ &= \quad OPT(I) + p_j(1 - \frac{1}{m}) \\ &\leq \quad OPT(I) + OPT(I)(1 - \frac{1}{m}) \\ &= \quad OPT(I)(2 - \frac{1}{m}). \end{aligned}$$

(The third step holds since $OPT(I)$ is at least p_j.)

Figure 2 depicts the optimal schedule for an instance consisting of $m(m-1)$ jobs of size 1 and one job of size m. If the greedy algorithm schedules the jobs of size 1 before the job of size m, then the makespan of the greedy schedule is $2m - 1$, while the optimal makespan is m. Therefore, the approximation ratio of the greedy algorithm is $2 - 1/m$.

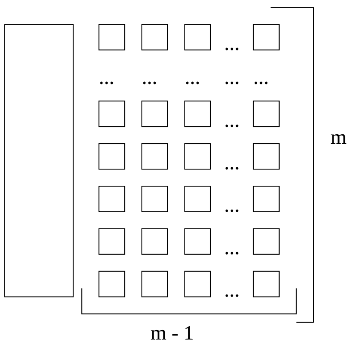

Figure 2: Worst case for greedy algorithm