Approximation Algorithms and Hardness of Approximation January 26, 2013

Lecture 3

Lecturer: Ola Svensson and Alantha Newman *Scribes: Chidambaram Annamalai*

1 Previous Lecture

In the previous lecture we saw examples of *greedy* algorithms that made locally optimal decisions at each step to arrive at a solution that wasn't too far from the optimal solution in the end. Specifically for the case of Set Cover we saw that this strategy leads to the best possible approximation algorithm we could hope for (unless $\mathsf{NP} \subset \mathsf{DTIME}(n^{\log\log n})$, which is very unlikely). In general, we also noted that greedy algorithms are usually easy to implement, and fast.

Today we will look at algorithms which can approximate the optimal solution as close as we want by trading off a sufficient quantity of time. For NP-Hard problems, this is the best we can hope for in terms of an approximation guarantee.

2 Trading off Time with Accuracy

Since we want to approximate the optimal solution of a maximization problem arbitrarily closely, we ask for a $(1-\epsilon)$-approximation algorithm A_ϵ for every choice of $\epsilon > 0$.

Definition 1 (PTAS) *A Polynomial Time Approximation Scheme (usually shortened to PTAS) is a family of algorithms $\{A_\epsilon\}_{\epsilon>0}$ such that for every $\epsilon > 0$, A_ϵ is a $(1-\epsilon)$-approximation algorithm.*

For a minimization problem we adjust the previous definition appropriately. Note that nothing in the defintion of a PTAS prevents the running time of the algorithm from depending on ϵ in a non-polynomial way. In fact, it is possible (and we look at such an example today) that the running time of A_ϵ could be $O(n^{(1/\epsilon)^{(1/\epsilon)}})$, which is exponential in the error term ϵ, but still polynomial in n for a fixed ϵ. A PTAS whose running time is polynomial in both n and $1/\epsilon$ is called a *Fully Polynomial Time Approximation Scheme or a FPTAS.*

3 0-1 Knapsack

In the knapsack problem we are given a set $S := \{1, 2, \ldots, n\}$ of n items, where each item has a size s_i and a profit p_i, and a capacity bound B. Our goal is to pack a subset of items whose total size is at most B such that the total profit from these items is maximized. Since the items are indivisible, this is called the 0-1 Knapsack problem.

3.1 Greedy Strategy

As a first try, let's try to formulate a greedy strategy for this problem. A greedy algorithm would try to maximize "bang for the buck" and always go for the item that maximizes the profit-to-size ratio (among all items that fit in the remaining capacity). However, such an algorithm performs arbitrarily poorly!

Lemma 2 *The greedy algorithm for* 0-1 *Knapsack has no guarantee*

Proof Consider an instance of knapsack with capacity 1 and the following two items: one with profit ϵ and size 2ϵ and the other with profit 1 and size 1. A greedy algorithm would pick the first item whereas the optimal solution is to pick the second. Since the ratio $\frac{2\epsilon}{1}$ is arbitrarily small, we cannot provide an approximation guarantee for this algorithm. ■

3.2 Dynamic Programming

Since our first attempt didn't quite work out, we will now try to solve the problem exactly using dynamic programming, and leverage this to build our FPTAS for the 0-1 Knapsack.

Dynamic Programming is a technique that applies to problems that exhibit the "optimal substructure" property. This means that for such problems, it is possible to extract the optimal solution for an instance by using only optimal solutions to smaller subproblems of the instance.

3.2.1 First Attempt

Consider some particular order of the items in S. To see the optimal substructure in case of knapsack, we define $M[i,b]$ as the maximum profit that we can achieve using only items from the set $\{1,2,\ldots,i\}$ and using capacity at most b. The solution of the base cases for this dynamic program $M[i,0]=0\ \forall i=0,\ldots,n$ and $M[0,b]=0\ \forall b=1,2,\ldots$ follow from the fact that we cannot achieve a non-zero profit when we are constrained to zero capacity or when we cannot use any items. To compute $M[i+1,b]$ we will use the recurrence

$$M[i+1,b]=\left\{\begin{array}{lc} \max(M[i,b-s_{i+1}]+p_{i+1},M[i,b]) & s_{i+1}\le b \\ M[i,b] & \text{else}\end{array}\right. .$$

The recurrence is based on the following idea: the optimal solution for $M[i+1,b]$ that maximizes the profit while only using items from $\{1,\ldots,i+1\}$ and using capacity at most b either uses the $i+1$-th item or it doesn't. In either case, we reduce the problem of determining $M[i+1,b]$ to one or more smaller subproblems.

The above dynamic programming formulation has approximately nB subproblems each of which we can solve in constant time using the recurrence relation combined with a bottom-up evaluation. So the total running time of the above dynamic program would be $O(nB)$. However, this is not polynomial in the length of the input since for a given 0-1 Knapsack instance that has a capacity bound B we can describe this using only at most $\lceil \log B\rceil$ bits (using an efficient encoding like binary encoding)[1].

Let's think about how we can use this pseudo-polynomial algorithm to build a FPTAS. Since, in general B could be very large (say, of the same order as 2^n) we would like to argue that we can "scale" the instance down by dividing the item sizes and B, solve the smaller problem exactly and then scale the problem back again. However, this method could encounter some problems because in the process of scaling up we could end up producing an infeasible solution due to rounding errors since the capacity bound B is a hard constraint.

3.2.2 Scaling Friendly Dynamic Program

So we turn the dynamic programming solution around and ask the question "what's the least capacity that I need in order to achieve a profit p using only items in $\{1,\ldots,i\}$". This leads to an alternate dynamic program for the problem, but one which is more suitable to scaling. Let us define $A[i,p]$ as the answer to our question (and we define it to be ∞ whenever this is not possible). As before we find that the base cases $A[0,p]=\infty$ if $p>0$ and $A[i,0]=0\ \forall i=0,1,\ldots,n$ since we cannot achieve profit $p>0$ with 0 items and we can achieve zero profit using zero capacity. Again, we branch into two cases based on the fact that the optimal solution to $A[i+1,p]$ either uses the $i+1$-th item or it doesn't. This leads to the following recurrence

$$A[i+1,p]=\left\{\begin{array}{lc} \max(A[i,p-p_{i+1}]+s_{i+1},A[i,p]) & p_{i+1}\le p \\ A[i,b] & \text{else}\end{array}\right. .$$

[1]In passing, we mention that algorithms whose running times become polynomial when the input is encoded in unary are called *pseudo-polynomial* algorithms. So we have just described a pseudo-polynomial algorithm for 0-1 Knapsack

Item	1	2	3	4
Size	1	2	4	1
Profit	1	2	4	2

Table 1: 0-1 Knapsack Instance with capacity $B = 5$

i, p	0	1	2	3	4	5	6	7	8	9
0	0									
1	0	1								
2	0	1	2	3						
3	0	1	2	3	4	5	6	7		
4	0	1	1	2	3	4	5	6	7	8

Table 2: Table displaying $A[i, p]$ (missing values are ∞)

Once we have computed $A[i, p]$ for all $i = 0, 1, \ldots, n$ and $p = 0, 1, \ldots, \sum_{i=1}^{n} p_i$ we can then find the optimal value by looking at the column of the table of values $A[i, p]$ with the highest profit such that there is still an entry that is no more than B.

For example we describe an instance of a 0-1 Knapsack problem in Table 3.2.2 and the corresponding values of $A[i, p]$ in Table 3.2.2. To see how table is computed let's look at a particular cell, say, $A[2, 3]$. Since Item 2 has profit of $2 \leq 3$ we look at the two possibilities: i) $A[1, 1] + 2$ and ii) $A[1, 3]$ corresponding to the two choices of either picking or not picking Item 2. Since $A[1, 3]$ is ∞, the minimum of the two is $A[1, 1] + 2 = 3$ which is the value of $A[2, 3]$. Similarly we compute the rest of the values.

The optimum for this instance is 6 since Table 3.2.2 tells us that it is possible to pack items totalling a profit of 6 within the capacity bound of $B = 5$.

In this case, we had to solve about $n \sum_{i=1}^{n} p_i$ subproblems each of which takes constant time to solve in a bottom-up fashion. Thus the running time of this algorithm is $O(n \sum_{i=1}^{n} p_i) = O(n^2 p_{\max})$, where $p_{\max}$ is the maximum profit over all items. Since we again have a pseudo-polynomial algorithm for the same problem it seems we have achieved little, but that's not true!

3.3 FPTAS for 0-1 Knapsack

Let's see how this formulation helps us. If in our instance, $p_{\max}$ is "small" ($\sim \mathsf{poly}(n)$) then we simply run the dynamic program to obtain the exact solution. If exactly one item has profit $\sim 2^n$ and others $O(n)$ then it is intuitively clear that we need not care much about items since $\frac{2^n}{2^n + (n-1)n} \approx 1$. The tells us that we can scale our instance down by dividing all the profits (since we care only about the higher order bits as we saw) and then solve this instance exactly to obtain a feasible solution that is not too bad. We can then scale this up without having to worry about losing feasibility since profit maximiziation is a soft constraint.

The algorithm that we are going to describe naturally follows this intuition of looking only at the higher order bits.

Algorithm 1 "Round Input"

```
Require: $\epsilon > 0$
1: Set $K := \frac{p_{\max}\epsilon}{n}$
2: Round profits by setting $p'_i = \lfloor \frac{p_i}{K} \rfloor$
3: Run the "scaling friendly" dynamic program with rounded profits to obtain a set of items $S$
4: return $S$
```

Lemma 3 *"Round Input" returns a feasible solution with profit at least* $(1 - \epsilon)\mathsf{OPT}$

Proof Let S be the subset of items returned by Algorithm 1. We denote by Profit and Profit$'$ the profit functions defined on subsets of items with the original profits p_i and the rounded profits p'_i respectively. Let the optimal solution be the set $O \subset \{1, \ldots, n\}$, so that $\mathsf{OPT} = \text{Profit}(O)$.

We now have the following chain of inequalities

$$\begin{aligned}
\text{Profit}(S) &\geq K\text{Profit}'(S) \quad \text{(due to scaling up)} \\
&\geq K\text{Profit}'(O) \quad \text{(since S is optimal for the rounded instance)} \\
&= K \sum_{i \in O} p'_i \\
&= \sum_{i \in O} K \lfloor \frac{p_i}{K} \rfloor \\
&\geq \sum_{i \in O} p_i - |O|K \quad \text{(accounting for rounding)} \\
&= \mathsf{OPT} - \epsilon \cdot \frac{|O|}{n} \frac{p_{\max}}{\mathsf{OPT}} \mathsf{OPT} \quad (*) \\
&\geq (1 - \epsilon)\mathsf{OPT},
\end{aligned}$$

where the (*) follows from the fact that $|O| \leq n$ and $\mathsf{OPT} \geq p_{\max}$ since we can at least achieve $p_{\max}$ in any instance. ■

In Algorithm 1, step 2 takes $O(n)$ time while step 4 takes $O(n^2 p'_{\max})$ time. Since $p'_{\max} \leq \frac{p_{\max}}{K} = \frac{n}{\epsilon}$, we have that the running time is $O(n^3 \frac{1}{\epsilon})$ which has a polynomial dependence on both n and $\frac{1}{\epsilon}$. Together with Lemma 3 this shows that Algorithm 1 is an FPTAS for 0-1 Knapsack.

4 Bin Packing

In the bin packing problem we are given a set $\{s_1, \ldots, s_n\}$ of n items $1 \geq s_1 \geq s_2 \ldots s_n \geq 0$ and our goal is to pack all the items into the least number of unit capacity bins. That is, in each bin the sum of the item sizes of items assigned to the bin should not exceed one.

There is a simple way to see that this problem cannot be easy using a reduction from Subset Sum which is a problem we know to be NP-Hard. We state the hardness of approximating bin packing as a theorem.

Theorem 4 *A $(\frac{3}{2} - \delta)$-approximation algorithm for Bin Packing, where $\delta > 0$, implies $P = NP$.*

Proof In Subset Sum we are given a set of integers $\{a_1, \ldots, a_n\}$ and are asked to decide whether there exists as subset A of $\{1, \ldots, n\}$ such that $\sum_{i \in A} a_i = \sum_{i \in \{1,\ldots,n\} \setminus A} a_i$. Suppose we had an oracle that is able to decide for any given instance of bin packing whether we required i) two bins or ii) three or more bins, then we can show that we can use it to solve Subset Sum. Given a Subset Sum instance $\mathcal{I}$ on the set $\{a_1, \ldots, a_n\}$, create an instance $\mathcal{J}$ of bin packing $\{s_1, \ldots, s_n\}$ by normalizing the a_i's such that their sum is 2 (i.e., $s_i := \frac{a_i}{(\sum_i a_i)/2}$). By construction, the solution to the Subset Sum instance $\mathcal{I}$ is "YES" if and only if the optimal solution to the bin packing instance $\mathcal{J}$ is exactly 2. The theorem now follows from the fact that a $(\frac{3}{2} - \delta)$-approximation to bin packing is exactly such an oracle. ■

So we see that it is hopeless to aim for a PTAS for bin packing. However, in the hardness of approximation proof from Theorem 4 we saw that the hardness lies really in deciding whether for a bin packing instance two bins suffice. From this, we might expect that bin packing is really only hard for small values of OPT, and we will see that this is indeed the case. For instance, there exists a bin packing algorithm that produces a solution of value no greater than $\frac{11}{9}\mathsf{OPT}(I) + 4$ for any instance I (despite the fact that $\frac{11}{9} < \frac{3}{2}$)[2].

Keeping this in mind, we will attempt to construct a PTAS for bin packing for "large" values of OPT, and we will call this an Asymptotic PTAS. Since we are looking at bin packing, we make the definition for a minimization problem.

Definition 5 (APTAS) *An Asymptotic PTAS (APTAS) is a family of algorithms $\{A_\epsilon\}_{\epsilon>0}$ such that for some constant $c > 0$ and for every $\epsilon > 0, A_\epsilon$ returns a solution of value no greater than $(1 + \epsilon)\mathsf{OPT}(I) + c$.*

As we saw earlier, we will try to use an exact algorithm for "bounded" instances to construct the APTAS for the general case. We will now see that if the number of different items and the size of the smallest item are bounded (from above and below respectively) by constants, then we have an exact polynomial time algorithm.

Theorem 6 *Suppose we have a bin packing instance I such that*

- *at most L items fit in a bin,*
- *there are at most M different item sizes.*

Then there exists an exact algorithm for bin packing with a running time of $O(n^{M^L})$

Proof The proof is simply to exhibit the brute-force algorithm that performs an exhaustive search over all possible configurations (ways of packing a unit capacity bin), and showing that there aren't too many such configurations that we need to consider. Let the number of feasible configurations be C. Since there are at most L different items that can fit in a bin, and each is of at most M different sizes, $C \leq M^L$. Let X_i denote the number of bins packed using configuration i (for $i = 1, \ldots, C$) in a valid solution. Since we need at most n bins in the worst case, the total number of solutions that we need to consider is upper bounded by the total number of non-negative integer solutions to the equation

$$X_1 + X_2 + \cdots + X_C \leq n, \tag{1}$$

which is $\binom{n+C}{n} \leq \binom{n+M^L}{n} = O(n^{M^L})$. So our algorithm is to enumerate all the non-negative integer solutions of equation (1) and for each solution look at the packing described by it and take the minimum over all feasible packings. ■

Having showed the previous lemma, it is now natural to try and reduce any given instance to one that doesn't have very small items and also doesn't have many different item sizes. Suppose we started with an instance $I = \{s_1, \ldots, s_n\}$ and reduce it to an instance I_R for some $\epsilon > 0$ such that

- I_R has all $s_i \geq \epsilon/2$ so that $L = \lfloor \frac{2}{\epsilon} \rfloor$ (ignore small items)
- I_R has at most $M = \lceil \frac{4}{\epsilon^2} \rceil$ different item sizes (grouping)

then we would like to relate $\mathsf{OPT}(I_R)$ to $\mathsf{OPT}(I)$. We do this in two steps, showing that each of the above operations (ignoring small items and grouping) does not affect the solution too much. In what follows, we use $\mathsf{SIZE}(I)$ to denote the sum $\sum_i s_i$, which is a simple lower bound on $\mathsf{OPT}(I)$.

[2]Perhaps even more surprising is the fact that nothing we know so far contradicts the possibility of the existence of an algorithm for bin packing with a guarantee of $\mathsf{OPT}(I) + 1$!

4.1 Ignoring Small Items

Suppose we divide the set of items I into "small" and "large" items so that $\mathsf{SMALL} := \{s_i | s_i < \epsilon/2\}$ and $\mathsf{LARGE} := \{s_i | s_i \geq \epsilon/2\}$. Now, we show that ignoring small items is justified in the following lemma.

Lemma 7 *Any packing of* LARGE *into l bins can be extended to a packing of I using at most* $\max\{l, (1+\epsilon)\mathsf{OPT} + 1\}$ *bins.*

Proof We start with the packing of items in LARGE into l bins. We now take all the items in SMALL and add them using First Fit creating extra bins when necessary. If FF did not use any extra bins then we only use l bins. If FF does use extra bins, then suppose after running FF, we have used $u + 1$ bins. We know that u of the bins must be at least $(1 - \epsilon/2)$ full, otherwise we would not have opened bin $u+1$. So, $\mathsf{SIZE}(I) \geq (1-\epsilon/2)u \implies u \leq (1+\epsilon)\mathsf{SIZE}(I)$. Since $\mathsf{OPT}(I)$ is at least $\mathsf{SIZE}(I)$, we have that $u + 1 \leq (1 + \epsilon)\mathsf{OPT} + 1$. ■

4.2 Grouping

We now describe the method we are going to use to reduce the number of different item sizes. A *grouping* of size k on an instance I produces an instance I' with at most $\frac{n}{k}$ different item sizes. The operation is as follows:

1. Sort the items in the order of decreasing size and define each continguous set of k items to belong to a particular group so that every group except possibly the last has exactly k items.

$$\underbrace{s_1 \geq s_2 \geq \cdots \geq s_k}_{G_1} \geq \underbrace{s_{k+1} \geq \cdots \geq s_{2k}}_{G_2} \geq \cdots \geq \underbrace{\cdots \geq \cdots \geq s_n}_{G_{\lceil \frac{n}{k} \rceil}}$$

2. Discard the first group G_1.

3. "Round up" each item to be the size of the largest item in its group.

The following lemma tells us how to use a packing of I' to create a packing for I using not too many more items in the process.

Lemma 8 *If I' is an instance obtained from I by performing a grouping operation of size k, then*

$$\mathsf{OPT}(I') \leq \mathsf{OPT}(I) \leq \mathsf{OPT}(I') + k.$$

Further, a packing of I' in B bins can be used to create a packing of I using no more than $B + k$ bins.

Proof Suppose we have the optimal packing of I using $\mathsf{OPT}(I)$ bins. Let the instance I' be represented by the items $\{s'_{k+1}, s'_{k+2}, \ldots, s'_n\}$. We can use the packing of instance I to create a packing for I' by placing item $s'_{k+1'}$ in place of s_1, s'_{k+2} in place of s_2, and so on. This shows that $\mathsf{OPT}(I') \leq \mathsf{OPT}(I)$.

Since each item of I' is at least as large as the corresponding item in I, we can use a packing of I' to pack all the items in I except those in G_1. We then pack the items in G_1 using no more than k bins showing the second inequality. Since this argument is constructive, we have the second claim. ■

4.3 An APTAS for Bin Packing

We now combine the results from Lemmas 7 and 8 giving the following algorithm for bin packing.

Algorithm 2 "APTAS for Bin Packing"

```
Require: $n$ items of sizes $s_1, \dots, s_n$ and $\epsilon > 0$
1: Set $\mathsf{SMALL} := \{s_i | s_i < \epsilon/2\}$ and $\mathsf{LARGE} := \{s_i | s_i \geq \epsilon/2\}$
2: Set $k := \lfloor \epsilon \mathsf{SIZE}(\mathsf{LARGE}) \rfloor$
3: Perform grouping of size $k$ on LARGE to get an instance $I_R$
4: Solve $I_R$ exactly using the algorithm from Theorem 6 with parameters $M = \frac{4}{\epsilon^2}, L = \frac{2}{\epsilon}$
5: Using the packing for $I_R$ produce a packing for LARGE as detailed in Lemma 8
6: Pack all the remaining items from SMALL using First Fit
7: return the final packing
```

From the grouping operation on the items in LARGE in step 3, we know that there are no more than $M = \frac{n_{\mathsf{LARGE}}}{k}$ different item sizes. By the setting of k in step 2, and the fact that $\mathsf{SIZE}(\mathsf{LARGE}) \geq \frac{\epsilon n_{\mathsf{LARGE}}}{2}$ it follows that $M \leq \frac{4}{\epsilon^2}$. This explains the correctness of step 4. Call the value of solution returned by Algorithm 2 on an instance I as $\mathsf{ALG}(I)$.

Since we packed the large items first, and then used First Fit for the small items,

$$\mathsf{ALG}(I) = \max\{l, (1+\epsilon)\mathsf{OPT}(I) + 1\}$$

using Lemma 7 where l is the number of bins that the algorithm uses to pack LARGE. But, using Lemma 8 we know that LARGE can be packed using $\mathsf{OPT}(\mathsf{LARGE}) + k$ bins using a grouping operation of size k. By our choice of k, we have $\mathsf{OPT}(\mathsf{LARGE}) + k \leq \mathsf{OPT}(\mathsf{LARGE}) + \epsilon\mathsf{SIZE}(\mathsf{LARGE}) \leq (1+\epsilon)\mathsf{OPT}(\mathsf{LARGE})$. Thus, we have:

$$\mathsf{ALG}(I) \leq \max\{(1+\epsilon)\mathsf{OPT}(\mathsf{LARGE}), (1+\epsilon)\mathsf{OPT}(I) + 1\} \leq (1+\epsilon)\mathsf{OPT}(I) + 1.$$

The running time of Algorithm 2 is dominated by step 4 where we solve the reduced instance I_R by brute force using parameters M and L. From Theorem 6 it follows that the running time of this APTAS for bin packing is $O(n^{M^L}) = O(n^{\frac{4}{\epsilon^2}^{\frac{2}{\epsilon}}})$.