15-854: Approximations Algorithms **Lecturer:** Anupam Gupta
Topic: Dynamic Programming **Date:** October 12, 2005
Scribe: Mihir Kedia

Note: Starred sections were not specifically covered in lecture - they serve to provide additional background and information.

10.1 Introduction

Only a brief explanation of dynamic programming is provided - see an undergraduate algorithms textbook for a more complete treatment. Dynamic programming is a concept used to efficiently solve optimization problems by caching subproblem solutions. More specifically, we can use this technique to solve problems that display the *optimal substructure* property - if a solution to a given problem includes the solution to a subproblem, then it includes the optimal solution to that problem.

There are a lot of NP-complete problems that are not vulnerable to dynamic programming. Our approach to writing approximation algorithms will be to construct relaxations of these problems problems that we can solve using dynamic programming.

10.2 Knapsack

Problem 10.2.1 (Knapsack) *As input, Knapsack takes a set of n items, each with profit p_i and size s_i, and a knapsack with size bound B (for simplicity we assume that all elements have $s_i < B$). Find a subset of items $I \subset [n]$ that maximizes $\sum_{i \in I} p_i$ subject to the constraint $\sum_{i \in I} s_i \leq B$.*

Knapsack is NP-hard through a reduction from the partition problem (see 10.5).

Using dynamic programming, we can get an exact solution for knapsack in time $O(poly(n, P_{max}) \times \log(nB))$. Unfortunately, this is not polynomial in the size of its representation - P_{max} is actually $\log P_{max}$ in the problem representation. Thus, we go back to our toolbox of approximation algorithms.

10.2.1 A 2-approximation algorithm

As a warmup, let's try a basic greedy algorithm:

Greedy Algorithm?

1. Sort items in non-increasing order of $\frac{P_i}{S_i}$.
2. Greedily pick items in above order.

Intuitively, we want the items with the most "bang for the buck". Unfortunately, this algorithm is arbitrarily bad. Consider the following input as a counterexample:

- An item with size 1 and profit 2
- An item with size B and profit B.

Our greedy algorithm will only pick the small item, making this a pretty bad approximation algorithm. Therefore, we make the following small adjustment to our greedy algorithm:

Greedy Algorithm Redux

1. Sort items in non-increasing order of $\frac{P_i}{S_i}$.
2. Greedily add items until we hit an item a_i that is too big. $\left(\sum_{k=1}^{i} s_i > B\right)$
3. Pick the better of $\{a_1, a_2, ..., \boxed{a\_k}\}$ and $\boxed{\{a\_\{k+1\}\}}$

At first glance, this seems to be a clumsy hack in order to deal with the above counterexample. However, it turns out that this algorithm has far better performance.

Theorem 10.2.2 *Greedy Algorithm Redux is a 2-approximation for the knapsack problem.*

Proof: We employed a greedy algorithm. Therefore we can say that if our solution is suboptimal, we must have some leftover space $B - S$ at the end. Imagine for a second that our algorithm was able to take a *fraction* of an item. Then, by adding $\frac{B-S}{s_{k+1}} p_{k+1}$ to our knapsack value, we would either match or exceed OPT (remember that OPT is unable to take fractional items). Therefore, either $\underline{\sum_{k=1}^{i-1} p_i \geq \frac{1}{2} OPT}$ or $p_{k+1} \geq \underline{\frac{B-S}{s_{k+1}} p_{k+1} \geq \frac{1}{2} OPT}$. ■

10.2.2 $1+\epsilon$ approximation

Let's look at our dynamic programming algorithm again. The main issue is that it it takes $O(poly(n, P_{max}))$ time. However, what if we simply scaled down the value of every item? The following algorithm is due to Kim and Ibarra [1].

FPTAS for knapsack

1. For some given error parameter $\epsilon > 0$, set $k = \left\lfloor \frac{n}{\epsilon} \right\rfloor$
2. For every item a_i, define $\widehat{p}_i = \left\lfloor \frac{p_i}{p_{max}} k \right\rfloor$
3. Run a dynamic programming algorithm (using $\{\widehat{p}_1, \widehat{p}_2, ..., \widehat{p}_n\}$ as our profit inputs) to get some optimal set $\widehat{S}$.
4. Output $\widehat{S}$.

We will explain the significance of the term FPTAS later. For now, let us prove a bound on the approximation abilities of the above algorithm.

Theorem 10.2.3 *FPTAS for knapsack is a $1 + \epsilon$-approximation for the knapsack problem.*

Proof: Let OPT be the optimal solution to our original knapsack instance. Since we obtain an exact solution to our transformed problem, we can make the following trivial claim.

$$\sum_{i \in \widehat{S}} \widehat{p_i} \geq \sum_{i \in OPT} \widehat{p_i} \tag{10.2.1}$$

$$\left(\frac{p_{max}}{k}\right) \sum_{i \in \widehat{S}} \widehat{p_i} \geq \left(\frac{p_{max}}{k}\right) \sum_{i \in OPT} \widehat{p_i} \tag{10.2.2}$$

Let us analyze both sides of 10.2.2 separately.

1. **FPTAS' profit.** If we scale up the profit in our reduced problem instance, we have to at least match the total profit achieved by the same set in the original set.

$$\sum_{i \in \widehat{S}} p_i \geq \sum_{i \in \widehat{S}} \left(\left\lfloor \frac{p_i}{p_{max}} k \right\rfloor \frac{p_{max}}{k} \right) \tag{10.2.3}$$

$$\sum_{i \in \widehat{S}} p_i \geq \left(\frac{p_{max}}{k}\right) \sum_{i \in \widehat{S}} \widehat{p_i} \tag{10.2.4}$$

$$\tag{10.2.5}$$

2. **OPT's profit**. We want to find a lower bound for $\left(\frac{p_{max}}{k}\right) \sum_{i \in OPT} \widehat{p_i}$ in terms of the optimal profit of the original problem instance $\left(\sum_{i \in OPT} p_i\right)$. The floor operator decreases its contents by at most one, so we can get rid of it to come up with the following series of inequalities.

$$\left(\frac{p_{max}}{k}\right) \sum_{i \in OPT} \left\lfloor \frac{p_i}{p_{max}} k \right\rfloor \geq \left(\frac{p_{max}}{k}\right) \left(\sum_{i \in OPT} \frac{p_i}{p_{max}} k - 1 \right) \tag{10.2.6}$$

$$\left(\frac{p_{max}}{k}\right) \sum_{i \in OPT} \left\lfloor \frac{p_i}{p_{max}} k \right\rfloor \geq \sum_{i \in OPT} p_i - \sum_{i \in OPT} \frac{p_{max}}{k} \tag{10.2.7}$$

$$\left(\frac{p_{max}}{k}\right) \sum_{i \in OPT} \left\lfloor \frac{p_i}{p_{max}} k \right\rfloor \geq \sum_{i \in OPT} p_i - \frac{n p_{max}}{k} \tag{10.2.8}$$

$$\left(\frac{p_{max}}{k}\right) \sum_{i \in OPT} \left\lfloor \frac{p_i}{p_{max}} k \right\rfloor \geq \sum_{i \in OPT} p_i - \epsilon p_{max} \tag{10.2.9}$$

$$\tag{10.2.10}$$

We know that $\epsilon p_{max} \leq \epsilon OPT$, leading us to our final inequality for OPT.

$$\left(\frac{p_{max}}{k}\right) \sum_{i \in OPT} \left\lfloor \frac{p_i}{p_{max}} k \right\rfloor \geq (1 - \epsilon) OPT \tag{10.2.11}$$

Combining 10.2.2, 10.2.4, and 10.2.11, we can arrive at the following conclusion.

$$\sum_{i \in \widehat{S}} p_i \geq (1-\epsilon)OPT \tag{10.2.12}$$

■

The running time for our algorithm is $O(poly(n, \frac{n}{\epsilon}, \log(nB))$. This leads us to our next topic.

10.3 PTAS and FPTAS

Note that in the above algorithm, we were able to specify the error parameter. We formalize this in the following definitions.

Definition 10.3.1 (Approximation Scheme) *An algorithm is an* approximation scheme *for a problem if, given some error parameter $\epsilon > 0$, it acts as a $O(1+\epsilon)$ approximation algorithm.*

Definition 10.3.2 (Polynomial Time Approximation Scheme) *An approximation scheme is a* polynomial time approximation scheme*, abbreviated PTAS, if for each* fixed $\epsilon > 0$*, the running time is bounded by a polynomial in the size of the problem instance.*

Intuitively, an approximation scheme allows us to specify the exact error in our approximation - the smaller the error parameter, the slower the approximation algorithm runs (otherwise we would be able to solve the problem in polynomial time). An interesting point to make is that we place no restrictions on the *rate* that the algorithm slows down at - all we require is that the algorithm run in polynomial time for any *fixed* $\epsilon > 0$. An extreme example would be an algorithm that runs in time $O(n^{(2^{(\frac{1}{\epsilon})})})$ - not the most useful of approximation schemes. We fix this by introducing the notion of a *fully polynomial time approximation scheme*, or FPTAS.

Definition 10.3.3 (Fully Polynomial Time Approximation Scheme) *An FPTAS is a PTAS with the modified constraint that the running time is bounded by a polynomial in the size of $\frac{1}{\epsilon}$ and the problem instance.*

Note that our above algorithm for knapsack is an FPTAS, since its running time is bounded by a polynomial over n and $\frac{1}{\epsilon}$.

10.3.1 More on FPTAS*

Assuming $P \neq NP$, an FPTAS is the best that we can hope for in dealing with NP-hard problems. Unfortunately, very few NP-hard problems admit an FPTAS. To explain this, we will have to revisit some basic NP theory.

In all of the problems we have considered, we have made the assumption that we encode all numbers in binary. We can also define the *unary size* of a problem instance - namely, the size of a problem instance where all the numbers are written in unary. Clearly, there are problems that run in polynomial time on the unary instance - our knapsack algorithm from above. There are also problems that continue to remain intractable on these unary instances. We make the following definitions.

Definition 10.3.4 (Strongly NP-hard) *A problem is strongly NP-hard if every problem in NP can be polynomially reduced to it in such a way that numbers in the reduced instance are always written in unary.*

Definition 10.3.5 (Pseudo-polynomial algorithm) *An algorithm for a problem is a pseudo-polynomial algorithm for that problem if its running time is bounded by a polynomial in its unary size.*

Most of the problems we have considered (as well as typical NP problems like 3-SAT) are strongly NP-hard. Unfortunately, by adding a very weak restriction we can show that if a problem is strongly NP-hard then it cannot admit an FTPAS (assuming $P \neq NP$.) The following theorem is due to Garey and Johnson [2].

Theorem 10.3.6 *Consider a integral-valued NP-hard minimization problem Π with the following restriction: Let B be a numerical bound which is polynomial in the unary size of a given NP-hard problem. On any instance of Π, the optimal solution is at most B.*

If Π admits an FPTAS, then it also admits a pseudo-polynomial algorithm.

Proof: Let us run our FPTAS on a problem instance with $\epsilon = \frac{1}{B}$. The returned value is at most the following.

$$(1+\epsilon)OPT < OPT + \epsilon B = OPT + 1 \tag{10.3.13}$$

In fact, the algorithm is required to return the optimal solution. Since the running time is polynomial in B (which is polynomial in the unary size of a problem), Π has a pseudo-polynomial algorithm. ∎

Corollary 10.3.7 *If an NP-hard problem with the restrictions given in theorem 10.3.6 is strongly NP-hard, then it does not admit an FPTAS (assuming $P \neq NP$.)*

Proof: If it admits an FPTAS then it admits a pseudo-polynomial algorithm - this contradicts the definition of strongly NP-hard. ∎

We now turn our attention to a problem that admits a PTAS, but does not admit an FPTAS.

10.4 Minimum Makespan Scheduling ($P||C_{max}$)

We have already seen this problem before.

Problem 10.4.1 (Minimum Makespan Scheduling) *Given n jobs, each with processing time p_i, and m identical machines, find an assignment of these jobs such that the completion time (makespan) is minimized.*

We need to be careful when doing an NP reduction. Reducing this problem from partition is trivial - by employing a slightly stronger problem, though, we can prove that Makespan is strongly NP-hard (see 10.5).

As a refresher, we used the concept of *List Scheduling* to develop a 2-approximation algorithm (which we were able to later refine to a 1.5-approximation). List scheduling simply refers to running a greedy algorithm - Order the jobs arbitrarily and continually place them on the least loaded machine.

We now want to develop a PTAS for this problem. The following PTAS is due to Hochbaum and Shmoys [3].

To do so, we will create a $(1+\epsilon)$ *relaxed decision procedure* for our problem.

Definition 10.4.2 ($(1+\epsilon)$ Relaxed Decision Procedure) *Given a bound T, an error parameter ϵ, and an instance of a the minimum makespan scheduling problem, a $(1+\epsilon)$* relaxed decision procedure *does the following.*

- *Will output **NO** if the minimum makespan is greater than T.*
- *If the minimum makespan is less than $(1+\epsilon)T$, will output the optimal schedule (i.e. **YES**).*

*Note that both of the above conditions may be true, in which case we do not care what our algorithm outputs (although it should be noted that our algorithm cannot always output **YES**, since that would solve the minimum makespan problem in polynomial time).*

How can we turn the above into an approximation algorithm? From our earlier work on a greedy algorithm for this problem (list scheduling algorithm), we have lower and upper bounds T_L and T_U such that $T_U = 2T_L$. Therefore, we use the following geometric binary search algorithm to arrive at a $1+\epsilon$ approximation.

Geometric Binary Search

1. Find the upper and lower bounds for our makespan. To find the lower bound T_L, take the higher of P_{max} and $\frac{1}{m}\sum_n p_i$ (the average amount of work on a machine). The upper bound T_U is twice that (by list scheduling proof).
2. Initialize our logarithmic bounds. More precisely, set $\log_2 T_1 = \log_2(T_L)$ and $\log_2 T_2 = \log_2(T_U)$
3. Continually do the following

   a. Set T to be the geometric midpoint of T_1 and T_2: $T = \frac{1}{2}(\log_2 T_1 + \log_2 T_2)$

   b. Run a $(1+\frac{\epsilon}{2})$ relaxed decision procedure on the problem. If **NO**, then increase T_1, otherwise decrease T_2.

   c. If $\frac{T_2(1+\frac{\epsilon}{2})}{T_1} \leq 1+\epsilon$, then terminate and output the $\widehat{S}$ associated with our current upper bound.

Proof: We want to prove that the above algorithm is correct, and that we make at most $\lg\frac{1}{\epsilon}$ iterations before our interval is small enough.

Binary search is intuitive. We know that our optimal makespan has to be between T_1 and T_2, so we can query points in that interval and shorten it appropriately. The important issue is determining when to terminate. We terminate when we know that we are within a factor of $1+\epsilon$, which is when the following inequality is true (note that this proves correctness).

$$\frac{T_2\left(1+\frac{\epsilon}{2}\right)}{T_1} \leq 1+\epsilon \tag{10.4.14}$$

$$\frac{T_2}{T_1} \leq \frac{1+\epsilon}{1+\frac{\epsilon}{2}} \tag{10.4.15}$$

We use geometric binary search with our initial interval equal to one (since the upper bound is twice the lower bound), so after t steps, our interval $\log T_2 - \log T_1 \leq 2^{-t}$. We want $\log \frac{T_2}{T_1} \leq \log 1+\epsilon$ to be less than 2^{-t}, so we can just set $t = \log \frac{1}{\epsilon}$. This proves the running time. ■

So now we simply need to develop our relaxed decision procedure. It turns out, however, that we will need *another* tool. Our other tool, which will be arbitrarily titled as *Coarse* $(1+\epsilon)$ *Relaxed Decision Procedure*, can solve all instances of this problem in which there are no jobs of size less than ϵT. Once we have that tool, we can use the following algorithm to solve the problem.

$(1+\epsilon)$ **Relaxed Decision Procedure**

1. Remove all jobs with processing time less than ϵT.
2. Run the *Coarse* $(1+\epsilon)$ *Relaxed Decision Procedure* on the remaining jobs. If it returns **NO**, then return **NO**
3. Use *list scheduling* to add back all the small jobs. If we cannot do this without violating our $(1+\epsilon)T$ bound, then return **NO**.

Proof: The second step is correct, since if we cannot stay within our bound without using small jobs, we certainly can't stay within it using those jobs.

The third step also works. If *list scheduling* fails, then there has to be some job that was not able to be placed. However, since we place it on the least loaded machine, that means that every machine has to have an amount of work greater than T (the maximum job size for these small jobs is ϵT). Thus, the minimum makespan must be higher than T. ■

All that needs to be done is to develop the final tool, and we'll be done. We want to develop a $(1+\epsilon)$ *Relaxed Decision Procedure* for all instances where the minimum job size is at least ϵT.

What are some of the benefits of our new instance? We now have a constant limit $(\frac{1}{\epsilon})$ on the number of jobs per machine. Moreover, akin to knapsack we can use the concept of *rounding*: this allows us to have a small number of possible sizes. With all of these amendments, it turns out that we can use a dynamic programming algorithm to solve this problem.

Coarse $(1+\epsilon)$ Relaxed Decision Procedure

1. Round each profit down to the nearest multiple of $\epsilon^2 T$
2. Use dynamic programming to exactly solve this problem.

We need to prove that this is a $1+\epsilon$ approximation, and exhibit how we can use dynamic programming to solve this problem.

Theorem 10.4.3 *The above algorithm achieves a $1+\epsilon$ approximation to its original input.*

Proof: There are at most $\frac{1}{\epsilon}$ jobs per machine, each of which is at most ϵ^2 more than its original value. This means that the returned makespan is at most $(1+\epsilon)OPT$. ■

Theorem 10.4.4 *We can solve the minimum makespan scheduling problem using dynamic programming*

Proof: Since ϵ is constant, both the maximum number of jobs per machine at the number of different job sizes are both constant. All we have to do is consider every possible set of jobs for a machine and use dynamic programming to cache optimal subproblem results.

More specifically, do the following. Assume that we have an instance where P represents the maximum number of jobs per machine and Q represents the number of different processing times. We define a vector $\vec{n} = \{n_1, n_2, \ldots, n_Q\}$ to enumerate all possible jobs, where the ith entry indicates that there are n_i jobs with processing time q_i in the problem instance. Let us finally create a set of vectors R to represent every single possible machine configuration (exclude the empty configuration). Note that $|R|$ is at most Q^P, since there are at most P jobs per machine. The minimum number of machines can be expressed using the following recurrence.

$$f(\vec{n}) = 1 + \min_{\vec{r} \in R} f(\vec{n} - \vec{r}) \tag{10.4.16}$$

We can reach a maximum depth of n, since we get rid of one job per iteration. By using dynamic programming to memoize results, we can achieve a polynomial time solution to the above recurrence.

■

That's all...

10.5 Addendum: Partition and 3-Partition*

This section serves to define the partition and 3-partition problems.

Problem 10.5.1 (Partition) *Given a set of n integers $\{a_1, a_2, ...a_n\}$ such that $\sum_{i=1}^{n} a_i = 2B$, find a subset of these integers that have a total of exactly B.*

This problem is *weakly* NP-hard - namely, there exists a pseudo-polynomial algorithm to solve this problem. We can use this problem to prove that knapsack is NP-complete.

Theorem 10.5.2 *Knapsack is an NP-complete problem.*

Proof: We restrict knapsack to partition. For every item i, set $s_i = p_i$ and $B = \frac{1}{2}\sum_{i=1} ns_i$. Since knapsack will attempt to maximize profit while staying within the bound of the knapsack, it will output a partition if one exists. ■

It should also be clear that we can use partition to prove that the minimum makespan scheduling problem is NP-complete by setting the number of machines to two. However, by employing a more difficult problem, we can prove that makespan is strongly NP-hard.

Problem 10.5.3 (3-partition) *Given a set of* $3n$ *integers* $\{a_1, a_2, ...a_n\}$ *and a bound* B *such that* $\sum_{i=1}^{3n} a_i = nB$*, partition these integers into* n *subsets of three elements each such that each group has a total of exactly* B*. Note that the requirement that there be exactly three integers in each group is usually expressed by requiring* $\frac{B}{4} < a_i < \frac{B}{2}$ *for all* i*.*

This problem is strongly NP-hard. Using it, we now can look at makespan.

Theorem 10.5.4 *Minimum Makespan Scheduling is an NP-complete problem.*

Proof: We restrict makespan to 3-partition. For some bound B, require that any item a_i have size $\frac{B}{4} < s_i < \frac{B}{2}$. Also require that $n = 3m$. ■

References

[1] O.H. Ibarra and C.E. Kim. Fast approximation algorithms for the knapsack and sum of subset problems. *Journal of the ACM*, 22:463-468, 1975.

[2] M.R. Garey and D.S. Johnson. Strong NP-completeness results: motivation, examples, and implications. *Journal of the ACM*, 25:499-508, 1978.

[3] D.S. Hochbaum and D.B. Shmoys. A polynomial approximation scheme for machine scheduling on uniform processors: using the dual approximation approach. *Siam Journal on Computing*, 17:539-551, 1988.