Advanced Algorithms

Lecture 2: Greedy and Dynamic Programming for Approximation Algorithms

Lecturer: Nathan Klein

1 Knapsack

In the knapsack problem, we have a knapsack that can carry an integer weight up to W. In front of us is a set of items $1, \dots, n$. Each item i has a positive integer weight w_i and a positive integer value v_i. Our goal is to take the highest value set of items which has total weight at most W. We'll assume that every item has weight at most W, as otherwise we can never take it.

It turns out this problem is NP-Hard. This lecture, we'll look at approximation algorithms for it to figure out where in the hierarchy of approximability it lies. We'll explore both a greedy algorithm and a dynamic programming algorithm. These algorithmic strategies are powerful for solving problems in P, so no surprise that they help us design approximation algorithms too.

1.1 A Greedy Algorithm for Knapsack

The natural thing to try is to repeatedly take the item which maximizes $\frac{v_i}{w_i}$, i.e., has the best weight to value ratio, and fits in the knapsack. Unfortunately this can be an arbitrarily bad approximation. Consider the 2 item instance where item 1 has $w_1 = 1, v_1 = 1 + \epsilon$, and item 2 has $w_2 = v_2 = W$. Then this algorithm will only take item 1, but you could have taken item 2, meaning greedy has an approximation ratio of no better than W. So this doesn't work.

It turns out, though, that a seemingly trivial modification fixes it. Let's say greedy took items $1, \dots, k$, where WLOG the items are sorted by their ratio $\frac{v_i}{w_i}$. Then, the fix is to either return $\{1, \dots, k\}$ or $\{k+1\}$, whichever is better. Let's call this "two-choice greedy."

Lemma 1.1. *Two-choice greedy is a 2-approximation.*

Proof. We need to argue that $v_{k+1} + v(\{1, \dots, k\}) \geq OPT$. If this is true, then we're done as it implies one of the solutions we consider is at least $\frac{1}{2}OPT$.

Let's consider a stronger OPT' that has a larger knapsack of weight $w(\{1, \dots, k+1\}) \geq W$. Then, it can get value $v_{k+1} + v(\{1, \dots, k\})$. But it cannot do any better, as this solution has the best possible ratio of value to weight, and completely fills the knapsack, implying it has the best possible value. Finally, clearly $OPT' \geq OPT$, which completes the proof. $\square$

1.2 A Dynamic Program for Knapsack

It turns out that there is a simple dynamic programming algorithm that solves knapsack exactly. So, P=NP. (OK, clearly not, but try to figure out *why* not while we discuss this.) Remember in dynamic programming the goal is to define a collection of subproblems so that any subproblem can be solved quickly if you know the solution to the others. Given a knapsack input with items $[n]$, we will have just n subproblems $A_1, \dots, A_n$.

Subproblem A_i will store the weights and values of possible solutions on items $1, \dots, i$. In particular, A_i will consist of a set of all pairs (w, v) for $w, v \in \mathbb{Z}_{\geq 1}$, $w \leq W$, for which there exists a solution of weight w and value v on items $1, \ldots, i$. Therefore, $|A_i| \leq WV$ where V is the value of the optimal solution.

Alright. So, A_0 just has one pair, $(0,0)$. Now, inductively assume we have A_{i-1}, and let's try to compute A_i. Well, it's easy, right? For every $(w,v) \in A_{i-1}$, we add (w,v) to A_i, and we also add $(w+w_i, v+v_i)$ if $w+w_i \leq W$.

Finally, to get the answer, we just look through every element in A_n, check which has the best value, and return that.

It's pretty intuitive, but let's prove that the algorithm is correct.

Lemma 1.2. *The above dynamic programming algorithm computes the value of the optimal solution.*

Proof. We will prove by induction that A_i contains all feasible pairs (w,v) for all possible sets of items in $\{1, \ldots, i\}$. This means that when we scan through A_n, we will return the optimal value.

The base case holds. So, assume that A_{i-1} contains all feasible pairs (w,v) for all possible sets of items in $\{1, \ldots, i-1\}$, and we'll prove the same for A_i. Let $S \subseteq \{1, \ldots, i\}$ be some set with $w(S) \leq W$. If $i \notin S$, then $(w(S), v(S)) \in A_{i-1}$ by assumption. If $j \in S$, then where $T = S \setminus \{j\}$, $(w(T), v(T)) \in A_{i-1}$ by assumption. Then in the algorithm we will see the pair $(w(T), v(T))$ and will add $(w(T \cup \{j\}), v(T \cup \{j\})) = (w(S), v(S))$ since $w(S) \leq W$. $\square$

Now, here's the big question: what's the running time of this algorithm? To compute A_i, we scan through A_{i-1}. We said the size of A_{i-1} is at most WV. So the running time is certainly at most nWV.

Homework Preview: On your homework you'll prove the running time can be improved to $O(n \cdot \min(W, V))$. You'll also show that a small modification of this algorithm can recover the solution itself, not just its value.

So why isn't P=NP? Well, it's because the running time of this algorithm is actually exponential in the input size, as the running time is linear in the *value* of the number W. But you can input a number of value n with $\log n$ bits.

Definition 1.3 (Pseudopolynomial Time)**.** *An algorithm runs in pseudopolynomial time if it runs in polynomial time when the numbers in the input are encoded in unary.*

And it turns out knapsack is only NP-hard when the numbers are exponentially large in the input size n.

1.3 Memoization

Remember memoization? Another way to do this using dynamic programming is to notice that to compute OPT on the first k elements, you only need to check two subproblems: either you took the kth element or you didn't. In other words, if $OPT(i,w)$ is OPT on the first i elements with bound w, then:

$$OPT(i,w) = \max(OPT(i-1,w), OPT(i-w, w-w_i) + v_i)$$

You can recursively compute the solution to $OPT = OPT(n, W)$ by memoizing: keeping a table of which subproblems have been computed so far, and only doing the computation if the subproblem hasn't been solved yet.

1.4 An FPTAS for Knapsack

Here we will show that you can get a FPTAS for knapsack, so a $1+\epsilon$ approximation for any $\epsilon > 0$ running in time polynomial in n and $\frac{1}{\epsilon}$. This algorithm is due to Ibarra and Kim from the 70s.

The intuition is that if we're OK with an approximate solution, the sizes of the input numbers don't have to be represented exactly. And from what you'll do on the homework, to get a polynomial time algorithm it's enough for the value of the optimal solution to be at most polynomial in n.

So, here's the algorithm. Create a new instance with weights w_i and values

$$v_i' = \left\lfloor \frac{n}{\epsilon} \cdot \frac{v_i}{\max_i v_i} \right\rfloor$$

for all items i. Then, run the dynamic programming algorithm above with running time $n \cdot \min(W, V)$ which also returns a solution. We will need to argue that the value of this solution (with respect to the original values) is at least $(1-\epsilon) \cdot OPT$.

First let's see the running time. We need to upper bound V in this instance. It's at most the value of the sum of all the items, which is at most

$$\sum_{i=1}^{n} \frac{n}{\epsilon} \cdot \frac{v_i}{\max_i v_i} \leq \frac{n^2}{\epsilon}$$

So, the algorithm is now truly polynomial time, not pseudo-polynomial time: it runs in $O(\frac{n^3}{\epsilon})$ steps.

Lemma 1.4. *Let S be the optimal solution returned on the set of values v'. Then $v(S) \geq (1-\epsilon) \cdot v(OPT)$.*

Proof. Let $\alpha = \frac{n}{\epsilon \cdot \max_i v_i}$ be the scaling factor we apply before the floor. Then:

$$\begin{aligned} v(S) &\geq \frac{1}{\alpha} \cdot v'(S) && \text{We rounded } v' \text{ down} \\ &\geq \frac{1}{\alpha} \cdot v'(OPT) && S \text{ is optimal for } v' \\ &\geq v(OPT) - \frac{n}{\alpha} \end{aligned}$$

The last inequality is because for each of the at most n elements in OPT, we lose at most 1 unit in v' by rounding down, and the term is multiplied by α. But now

$$\frac{n}{\alpha} = \epsilon \max_i v_i \geq \epsilon OPT$$

Since every element fits in the knapsack. But this implies $v(OPT) - \frac{n}{\alpha} \geq (1-\epsilon)OPT$, completing the proof. □