CS 31: Algorithms (Spring 2019): Lecture 7

Date:

Topic: Dynamic Programming 2: The Knapsack Problem

Disclaimer: These notes have not gone through scrutiny and in all probability contain errors. Please notify errors on Piazza/by email to deeparnab@dartmouth.edu.

In this note we see a cousin of the Subset Sum problem done last lecture. It is the first example of an *optimization* problem. The Subset Sum problem was a *decision* problem, in that, the output was YES or NO (ok, so if YES we also wanted the subset). In optimization problems, the question is not whether a feasible solution exists, but more of among all *candidate feasible solutions* can you choose one which is *best* in a certain metric.

1 Knapsack Problem.

KNAPSACK
Input: n items; item j has profit p_j and weight w_j. A knapsack of capacity B. All of these are positive integers.
Output: Find the subset $S \subseteq \{1, 2, \ldots, n\}$ which maximizes $\sum_{j \in S} p_j$ and "fits" in the knapsack; that is, $\sum_{j \in S} w_j \le B$.

Note the question "does there exist a subset which fits in the knapsack?" is trivial to answer. Yes, there is: the *empty set*. The interesting part is to figure out which among all candidate subsets, gives the largest profit. As in Subset Sum, the brute-force approach of going over all possible subsets that fit in the knapsack and choosing the best, is a time consuming affair. We want to do better via dynamic programming.

As in the Subset Sum case, let us fix an instance I of Knapsack $((p_1, w_1), \ldots, (p_n, w_n); B)$. Let us abstractly consider an optimal solution $S \subseteq \{1, \ldots, n\}$ for this problem (the subset marks the indices of the items picked). Can we break this S up into solutionettes for smaller instances of Knapsack? We will proceed *exactly* like in Subset Sum.

Let us focus on the "last" item n and ask whether it is in S or not.

- If it *is* in S, then I claim $S_1 = S \setminus n$ is an *optimal* (max profit) solution to a smaller instance of Knapsack. Can you see which one? It is indeed that $I_1 = ((p_1, w_1), (p_2, w_2), \ldots, (p_{n-1}, w_{n-1}); B - w_n)$. Why is S_1 the best solution in I_1? Well, if there was something better, then adding the nth item to that solution would give a better solution than S to the original instance I.

- If the nth item *is not* in S, then again S itself is the optimal (max-profit) solution to the smaller instance $I_2 = ((p_1, w_1), (p_2, w_2), \ldots, (p_{n-1}, w_{n-1}); B)$ of Knapsack. Again, if not, then a better solution for this smaller instance would be a better solution for the original instance.

One can also argue the *vice-versa* direction: given optimal solution to both I_1 and I_2, can we find the optimal solution to I? Can you guess how to do it? We will take the solution to I_1 and *add* the profit p_n of the nth item, and compare it to I_2 (when we don't add the nth item). And take the one that is better (gives more profit). The best of these two will be the best solution for I.

All the above discussion, again, is the thoughts going in our head which lead us towards the rigorous solution to the dynamic programming problem. At this point, we should perhaps draw the tree diagram for the recursive structure of the problem (as in Figure 1 in the previous lecture notes), and we will see as in Subset Sum, they arrange up in a grid. There are two parameters of interest: m, denoting the first m items, and b, the available size in the knapsack. After we do all this, it is time to venture on to the 7-fold path we laid down last time.

Before we do so, let me introduce another piece of notation which is going to be very useful for *arguing* about *optimization* problems. It is the notion of Cand which captures the collection of *candidate feasible* solutions to the smaller instance one is considering. For the Knapsack problem, since we know that m and b are the parameters of interest, we define the following:

$$\mathsf{Cand}(m,b) : \text{all possible subsets of } \{1,2,\dots,m\} \text{ of items with total weight is} \le b.$$

In English, $\mathsf{Cand}(m,b)$ are the candidate feasible solutions to the instance $((p_1,w_1),\dots,(p_m,w_m);b)$. And by definition, the *best* (maximum profit) solution is the one giving the maximum value. For writing our recurrence, it will be this value that will be most important, and this is going to be the part of our definition.

We will write a recurrence for $F(m,b)$ which is the maximum profit subset in $\mathsf{Cand}(m,b)$.

1. ***Definition:*** For any $0 \le m \le n$ and $0 \le b \le B$, let $\mathsf{Cand}(m,b)$ be all subsets $S \subseteq \{1,\dots,m\}$ which fit in a knapsack of capacity b, that is, $\sum_{j\in S} w_j \le b$. Define

$$F(m,b) = \max_{S\in\mathsf{Cand}(m,b)} \sum_{j\in S} p_j$$

   We use shorthands $p(S) = \sum_{j\in S} p_j$ and $w(S) = \sum_{j\in S} w_j$ for brevity. We are interested in $F(n,B)$.

2. ***Base Cases:***
   - $F(0,b) = 0$ for all $0 \le b \le B$; an empty set gives profit 0.
   - $F(m,0) = 0$ for all $0 \le m \le n$; an empty set gives profit 0.

3. ***Recursive Formulation:*** As can be deduced from the discussion above, we assert for all $m \ge 1, b \ge 1$:

$$F(m,b) = \max\left(\ F(m-1,b),\ F(m-1,b-w_m)+p_m\ \right)$$

4. ***Formal Proof:*** As in Subset Sum, we need to show an equality. We do so by proving the two inequalities. In what follows, we first show that the left hand side (LHS) is ≤ the right hand side (RHS). Subsequently, we show LHS ≥ RHS. This proves LHS = RHS. We will see that the set Cand will be useful in proving this.

   (≤): Let S be the subset in $\mathsf{Cand}(m,b)$ such that $F(m,b) = p(S)$.
   Case 1: S doesn't contain item m. Then $S \in \mathsf{Cand}(m-1,b)$ and so $F(m-1,b) \ge p(S) = F(m,b)$, since $F(m-1,b)$ is the *maximum* over all sets in $\mathsf{Cand}(m-1,b)$.
   Case 2: S contains item m. Then $S \setminus m$ lies in $\mathsf{Cand}(m-1,b-w_m)$ and $p(S\setminus m) = p(S)-p_m = F(m,b)-p_m$. Thus, $F(m-1,b-w_m) \ge F(m,b)-p_m$, since $F(m-1,b-w_m)$ is the *maximum* over all sets in $\mathsf{Cand}(m-1,b-w_m)$.

   (≥): Let S be the subset in $\mathsf{Cand}(m-1,b)$ such that $p(S) = F(m-1,b)$. Observe S also lies in $\mathsf{Cand}(m,b)$. Thus, $F(m,b) \ge p(S) = F(m-1,b)$ since $F(m,b)$ is the *maximum* over all sets in $\mathsf{Cand}(m,b)$.
   Similarly, let S be the subset in $\mathsf{Cand}(m-1,b-w_m)$ such that $p(S) = F(m-1,b-w_m)$. Form $S' = S+m$. Note that $S \in \mathsf{Cand}(m,b)$ since $w(S') \le b$, and $p(S') = F(m-1,b-w_m)+p_m$. Thus, $F(m,b) \ge F(m-1,b-w_m)+p_m$.

5. ***Pseudocode for computing*** $F[n,B]$ ***and recovery pseudocode:*** The pseudocode is one formed, as in Subset Sum, by the smart recursion idea on the above recurrence equality. The recovery process is also similar.

procedure KNAPSACK(B,$(p_1,w_1),\cdots,(p_n,w_n)$):
    ▷ *Returns the subset of items of type $1,\ldots,n$ which fits in knapsack of capacity B and gives maximum profit.*
    Allocate space $F[0:n,0:B]$
    $F[0,b] \leftarrow 0$ for all $0 \le b \le B$ ▷ *Base Case*
    $F[m,0] = 0$ for all $0 \le m \le n$. ▷ *Base Case*
    for $1 \le m \le n$ **do**:
        for $1 \le b \le B$ **do**:
            if $b - w_m \ge 0$ **then** :
                $F[m,b] \leftarrow \max(F[m-1,b], F[m-1,b-w_m]+p_m)$
                ▷ *Note $F[m-1,b-w_m]$ is set before $F[m,b]$ in this ordering.*
            else: ▷ *Implicitly, in this case $F[m-1,b-w_m] = -\infty$*
                $F[m,b] \leftarrow F[m-1,b]$
    ▷ *$F[n,B]$ now contains the value of the optimal subset*
    ▷ *Below we show the recovery pseudocode*

    $m \leftarrow n$; $b \leftarrow B$; $S \leftarrow \emptyset$.
    ▷ *Invariant: $\sum_{j\in S} w_j + b \le B$ and $F[m,b] + \sum_{j\in S} p_j = F[n,B]$*
    while $m > 0$ **do**:
        if $F[m,b] = F[m-1,b]$ **then**:
            $m \leftarrow m-1$
        else: ▷ *We know $F[m,b] = F[m,b-w_m]+p_m$.*
            $S \leftarrow S + m$
            $b \leftarrow b - w_m$.
            $m \leftarrow m-1$
    return S

Note that in the recovery the invariant always holds and at the end since $F[0,k] = 0$, we have $p(S) = F[n,B]$.

6. ***Running time and space*** The above pseudocode take $O(nB)$ time and space where n is the number of items.