CIS 320: Algorithms September 29, 2021

Lecture 8

Lecturer: Aaron Roth *Scribe: Aaron Roth*

Dynamic Programming II: Knapsack

In the last lecture, we solved our first problem with "dynamic programming". Recall that this is one of several terrible names for the otherwise simple process of "writing down a recursive algorithm and caching the solutions to sub-problems so that you don't have to wastefully compute them multiple times". Weighted Interval Scheduling was a "one dimensional" problem in the sense that the sub-problems were specified by a single parameter, and when unrolling the recurrance, we found we only had to fill in the entries for a single 1-dimensional array. But there is no reason that dynamic programming has to be 1-dimensional in this sense: today we'll see a problem that is "two dimensional".

Definition 1 (The Knapsack Problem) *An instance $I = (\{s_i, v_i\}_{i=1}^n, C)$ of the knapsack problem is given by a non-negative integer* capacity C *together with a collection of n items $i \in \{1, \ldots, n\}$, each of which has a non-negative integer* size s_i *and* value v_i*. A subset $S \subseteq [n]$ of the items is* feasible *if $\sum_{i \in S} s_i \leq C$, and it has value $v(S) = \sum_{i \in S} v_i$. The goal of the Knapsack problem is to find the feasible subset of items of maximal value.*

Remember that the first step in coming up with a dynamic programming algorithm is coming up with a recursion. It often helps to work backwards. So lets think. Suppose S^* is the optimal solution to a knapsack instance I. Is $n \in S^*$? There are two possibilities. Either:

1. $n \notin S^*$. In this case, the optimal solution to I is the same as the optimal solution to $I' = (\{s_i, v_i\}_{i=1}^{n-1}, C)$. (Otherwise we could improve the value of the solution while maintaining feasibility by picking $\text{OPT}(S')$... *Or*:

2. $n \in S^*$. In this case, after accounting for item n, we have remaining capacity $C - s_n$. The optimal solution should fill this capacity optimally with the remaining items (else we'd have an improvement) — so the optimal solution is the union of $\{n\}$ together with the optimal solution to $I' = (\{s_i, v_i\}_{i=1}^{n-1}, C - w_n)$.

This suggests that we will need to solve sub-problems that vary along *two* dimensions: how many items are in the instance, *and* the capacity. Given an instance I, let us therefore define the following quantity:

$$\text{OPT}(k, C') = \max_{S \subseteq [1,\ldots,k]: \sum_{i \in S} s_i \leq C'} \sum_{i \in S} v_i$$

which is the value of the optimal solution to the sub-instance on items $1, \ldots, k$ with capacity C'. Translating the above reasoning into this notation, we have that:

$$\text{OPT}(n, C) = \max\left(\text{OPT}(n-1, C), v_n + \text{OPT}(n-1, C - s_n)\right)$$

With our recursion in hand, the hard part is done: we can now write down a recursive algorithm. Observe that once again, the naive recursion would have exponential running time, since the depth of the recursion tree is n and it has branching factor 2. But we will cache our intermediate results (i.e. "memoize") to avoid duplicating effort: We will now have a two dimensional array indexed as $M[k, C]$, corresponding to each of our sub-problems.

Algorithm 1 M-Compute-OPT(k, C)

```
if k = 0 or C ≤ 0 then
  Return 0.
else if M[k, C] is not empty then
  Return M[k, C]
else
  Let M[k, C] = max(M-Compute-OPT(k − 1, C), v_k + M-Compute-OPT(k − 1, C − s_k))
  Return M[k, C]
end if
```

Theorem 2 *On any instance I, M-Compute-OPT(n, C) returns the optimal solution in time $O(n \cdot C)$.*

Proof The correctness of the algorithm follows by an induction on $k+C$ together with the recurrence we derived above. In the base cases in which either $k = 0$ or $C \leq 0$, the algorithm returns 0 which is correct.

In the remaining case, we may assume by our inductive assumption that M-Compute-OPT$(k-1, C) =$ OPT$(k-1, C)$ and M-Compute-OPT$(k-1, C-s_n)$ returns OPT$(k-1, C-s_n)$. It then follows from our recurrance above that the algorithm returns the correct solution:

$$\text{OPT}(k, C) = \max\left(\text{OPT}(k-1, C), v_k + \text{OPT}(k-1, C-s_k)\right).$$

To bound the running time, we observe that the algorithm takes $O(1)$ time per recursive call. It makes 2 recursive calls for each entry of M (the first time it is accessed), and the total number of entries in M is $n \cdot C$. ■

Just as in last lecture, once we have used this algorithm to compute the value of OPT(n, C), from the matrix M that is produced, we can trace the recursion backwards through it and recover the optimal solution in $O(n)$ time:

Algorithm 2 Extract-Solution(k, C)

```
if k = 0 then
  Return ∅.
else
  if v_k + M[k − 1, C − s_k] ≥ M[k − 1, C] then
    Return {k} ∪ Extract-Solution(k − 1, C − s_k)
  else
    Return Extract-Solution(k − 1, C)
  end if
end if
```

We get linear run time because at most one recursive call is made for each of the n values of k, and the algorithm does a constant amount of work per call.

Finally, just as before, we can unroll the recursion and give an iterative version of our dynamic program, which has practical advantages. We now need to fill out a two dimensional array: once again, since the recursive calls always reference values of M indexed at $k-1$, if we fill out the table so that every entry at level k is completed before we move to level $k+1$, we will always have the information we need to proceed.

Algorithm 3 Iterative-Compute-OPT(I)

Initialize an $n \times C$ array M such that $M[0, c] = 0$ and $M[k, 0] = 0$ for all i, j.
for $k = 1$ to n **do**
  for $c = 1$ to C **do**
    Let $M[k, c] = \max(M[k-1, c], v_k + M[k-1, c - s_k])$
  end for
end for
Return $M[n, C]$.

Once again, this algorithm is doing exactly the same thing as the recursive algorithm, but now the $O(nC)$ run-time is a little more transparent.

Observe that what we have come up with is not technically a polynomial time algorithm, because it runs in time that is polynomial (linear) in C, rather than polynomial in the number of bits ($\log C$) needed to represent C. This is sometimes called a "pseudo-polyomial time" algorithm. Nevertheless, this can be a useful algorithm in settings in which C is not enormous. A real polynomial time algorithm for Knapsack is extremely unlikely, as it is an NP-hard problem.