CSE 347 Lecture 9

Dynamic Programming II: knapsack & string matching

1. A pseudo-polynomial-time algorithm for the knapsack problem
2. A dynamic programming algorithm for the weighted sequence alignment problem

---

Today, we'll see some more sophisticated uses of dynamic programming, illustrating the scope of its potential.

The first problem we consider is called the Knapsack problem: we have n items, each of which has a specified weight w_i and value v_i. We are given a maximum weight W we can carry, and we wish to find a subset of the items of total weight at most W of maximum total value.

We first observe that natural greedy approaches fail, even for the special case where the value of each item is equal to its weight — e.g. if we sort by decreasing weight, then in $\{\frac{W}{2}+1, \frac{W}{2}, \frac{W}{2}\}$, the greedy choice is $\{\frac{W}{2}+1\}$, whereas $\{\frac{W}{2}, \frac{W}{2}\}$ (of total weight/value W) is optimal; similarly, if we sort by increasing weight, then in $\{1, \frac{W}{2}, \frac{W}{2}\}$, the greedy choice is $\{1, \frac{W}{2}\}$ (of value $\frac{W}{2}+1$), but $\{\frac{W}{2}, \frac{W}{2}\}$ is again

optimal. So, we need to do something more clever.

Since this is a lecture on dynamic programming, naturally that's what we'll use. The tricky part is coming up with the subproblems. What would you try? The natural first choice, inspired by our weighted scheduling problem, is to consider the first j items in increasing weight. But, just because we take the ith item doesn't mean that we can't take the $(i-1)^{th}$, nor does it mean that we can use the optimal subset of the first $i-1$ — consider again $\{1, \frac{W}{2}, \frac{W}{2}\}$

The additional constraint imposed by including an item is reflected in the remaining weight. That is: our subproblems will be specified by both the items remaining and the weight remaining (which we assume is an integer). Thus, for n items and a bound of W on the weight, we will have nW subproblems (for weight $1, 2, \dots, W$, and the first j items, $j = 1, \dots, n$).

Now we can easily express, for example, the optimal value via the recurrence

$$opt(j, W) = \begin{cases} opt(j-1, W) & \text{if } w_j > W \quad (j^{th} \text{ item too big}) \\ \max\{opt(j-1, w), v_j + opt(j-1, W-w_j)\} & \text{o.w.} \end{cases}$$

with base case $opt(0, w) = 0$.

As with our weighted scheduling problem from last time, we'll be able to easily recover an optimal set of items from the table of values.

This recurrence only requires the solution to sub-problems on the first $j-1$ items, so we will proceed, for $j=0,1,\ldots,n$, to fill in the solutions for all $w \le W$:

Input: $\{(w_1,v_1),\ldots,(w_n,v_n)\}$, integer weight bound W

Initialize a $n \times W$ array M with $M[0,w]=0$ for all w.

For $j=1,2,\ldots,n$
  For $i=0,\ldots,W$
    If $i < w_j$ then put $M[j,i] \leftarrow M[j-1,i]$
    Else if $M[j-1,i] \ge v_j + M[j-1,i-w_j]$
      Put $M[j,i] \leftarrow M[j-1,i]$
    Else put $M[j,i] \leftarrow v_j + M[j-1,i-w_j]$

What is the running time of this algorithm? $O(nW)$.

Theorem The algorithm terminates with $M[j,i]$ containing the maximum value of a subset of $\{(w_1,v_1),\ldots,(w_j,v_j)\}$ of total weight at most i.

Proof? By induction on j. Base, $j=0$ — no items have 0 value. Induction step: If $w_j > i$, then no solution can use the jth item, so the optimal subset is the same as for $\{(w_1,v_1),\ldots,(w_{j-1},v_{j-1})\}$ with bound i, which by IH has its value in $M[j-1,i]$ (and this is what we put in $M[j,i]$. Otherwise, either there is an optimal assignment that does not use the jth item — with value contained in $M[j-1,i]$ by IH — or else we use the jth item. In the latter case, the remaining capacity is $i-w_j$, and we can do no better than to choose an optimal subset of the first $j-1$ items with this

capacity, which has its value stored in $M[j-1, i-w_j]$ by IH. The total value of the solution is then $v_j + M[j-1, i-w_j]$. So, the optimum value is whichever of these two values is larger, and that is what we take for $M[j,i]$ ▨

This algorithm runs in time polynomial in the weight bound W (and n), not the number of digits in W. Such an algorithm is said to be a "pseudo polynomial" time algorithm. It's feasible only so long as W is small.

To recover the set of items, we can use the following

Initialize $j = n$, $i = W$, $A = \emptyset$
While $j, i > 0$
  If $M[j,i] \neq M[j-1,i]$, then put $A \leftarrow A \cup \{j\}$, $i \leftarrow i - w_j$
  Put $j \leftarrow j-1$

Observe: this takes time $O(n)$, and since we only ever set $M[j,i]$ to a value other than $M[j-1,i]$ when we must take the jth item, it follows by induction on j that we will include an optimal subset of the first j items of total weight at most i in A after an iteration with i, j as our indices.

Sequence Alignment

Next, we'll consider a problem that arises in spelling correction and in computational biology. Suppose we are given a string like 'ocurrance' [sic] and we wish to find the nearest match—here, "occurrence". We define the distance between the two strings to be

the minimum number of deletions, insertions, and substitutions to transform one to the other. Or, more generally, we may declare each of these operations to have a cost — so, while occurrence costs one insertion and one substitution, occurr-ence costs two insertions and a deletion, which we could conceivably prefer. We will suppose that insertions & deletions cost $\delta > 0$, and for a pair of symbols p & q in our alphabet, substituting q for p costs α_{pq} — which, we assume is zero if $p=q$ and strictly positive otherwise. So, we prefer the first alignment of 'ocurrance' and "occurrence" to the second if and only if $\delta + \alpha_{ae} < 3\delta$.

We'll denote our inputs by $X = x_1 x_2 \cdots x_m$ and $Y = y_1 y_2 \cdots y_n$
We'll use a recurrence based on the following observation

Lemma: If X is transformed to Y then either
(1) y_n is substituted for x_m, (2) x_m is deleted, or
(3) y_n is inserted.

Proof: If neither x_m is deleted nor is y_n inserted, the first string ends with x_m and the second ends with y_n, so x_n was transformed into y_n by a substitution ∎

Notice that in the first case, we are left with a subproblem of transforming $x_1 \cdots x_{m-1}$ into $y_1 \cdots y_{n-1}$; in the second, we must transform $x_1 \cdots x_{m-1}$ into $y_1 \cdots y_n$; and, in the third, we transform $x_1 \cdots x_m$ into $y_1 \cdots y_{n-1}$. This suggests a set of $m \cdot n$ subproblems, aligning the first i

symbols of X with the first j symbols of Y.

What would the dynamic programming algorithm for computing the value be?

Input $X = x_1 \cdots x_m$, $Y = y_1 \cdots y_n$, costs $\delta > 0$, α_{pq} for all pairs p, q.

Initialize a $m \times n$ table M with $M[i,0] = \delta \cdot i$, $M[0,j] = \delta \cdot j$

For $i = 1, \ldots, m$

  For $j = 1, \ldots, n$

    Put $M[i,j] \leftarrow \min\{\alpha_{x_i y_j} + M[i-1, j-1], \delta + M[i-1, j], \delta + M[i, j-1]\}$

From the table of costs M, we can recover the optimal alignment in $O(m+n)$ time just as before. Observe that this algorithm runs in time $O(mn)$ time overall.

Theorem $M[i,j]$ contains the minimum cost of transforming $x_1 \cdots x_i$ to $y_1 \cdots y_j$.

Proof: By induction on $i+j$. Base, $i = 0$ or $j = 0$: If either X or Y is empty, then it takes, respectively, j insertions or i deletions, which cost, respectively, $\delta \cdot j$ and $\delta \cdot i$, which are the entries in $M[0,j]$ and $M[i,0]$. ✓

Induction step: By our earlier lemma, either y_j is substituted for x_i, and $x_1 \cdots x_{i-1}$ is (somehow) transformed into $y_1 \cdots y_{j-1}$, or x_i is deleted and $x_1 \cdots x_{i-1}$ is transformed into $y_1 \cdots y_j$, or y_j is inserted and $x_1 \cdots x_i$ is transformed into $y_1 \cdots y_{j-1}$. By IH, the first cost is $\alpha_{x_i y_j} + M[i-1, j-1]$, the second cost is $\delta + M[i-1, j]$, and the third cost is $\delta + M[i, j-1]$ (Note: both are at most $i+j-1$) Thus, the minimum cost is the smallest of these, which is stored in $M[i,j]$ ▨