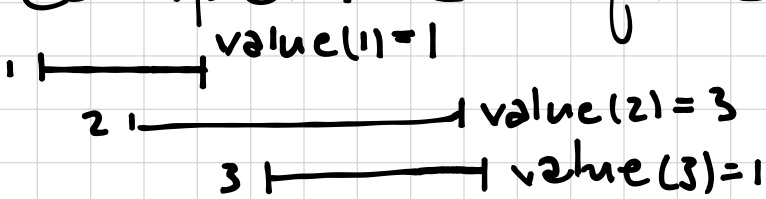
CSE 347 Lecture 8

Dynamic Programming I: Weighted interval scheduling

1. The weighted interval scheduling problem
2. An inefficient recursive algorithm
3. Dynamic programming: memoizing + bounding

In divide-and-conquer, we saw a technique for giving faster algorithms for problems that already had feasible algorithms. This is in contrast to the greedy algorithms we found for problems like interval scheduling and minimum spanning tree, that did not have obvious, feasible algorithms. Today, we'll see a kind of algorithm that can solve many more problems that do not have obvious feasible algorithms.

We return to the scheduling problem, in which each request has a value (a weight), and we wish to find a compatible set of requests of the greatest possible total value. We first observe that the greedy algorithm that solved the problem when every request was of equal value is no longer correct — we were repeatedly choosing the compatible request that finished earliest.



Earliest finish chooses 1, then 3
Total value: value(1) + value(3) = 2

Choosing 2 alone gives value(2) = 3 > 2.

(Recall that a set of requests was compatible if none of them overlap. So, once our greedy algorithm chooses 1, it can't choose 2.)

Let's start by considering an inefficient algorithm that is obviously correct: we consider all possible subsets of requests, disregard those that are incompatible, and take one of maximum weight. We can write this as a recursive algorithm

Find Opt

Input: a set of requests with weights, $R=\{(r_1,w_1),\dots,(r_n,w_n)\}$

If $R=\phi$, return $(\phi,0)$

Put $(A_0,W_0)\leftarrow$ FindOpt$(R-\{(r_n,w_n)\})$

Put $(A_1,W_1)\leftarrow$ FindOpt$(\{(r,w)\in R: r\cap r_n=\phi\})$

If $W_0>W_1+w_n$, return (A_0,W_0)

Else return $(A_1\cup\{r_n\},W_1+w_n)$

This doesn't seem great so far, but here is the first key idea: let's suppose that $r_1,\dots,r_n$ are sorted by increasing finish times. That way, if r_n conflicts with any request r_k, it must also conflict with $r_{k+1}, r_{k+2},\dots,r_{n-1}$. Now, how many distinct recursive calls does Find Opt make? At most $n-1$, since each recursive call is made on some prefix of the list $(r_1,w_1)\cdots(r_n,w_n)$, and there are $n-1$ prefixes. We'll let $p(k)$ = the index before the first conflict with r_k.

This brings us to the second key idea: instead of re-evaluating these recursive calls, we can cache

or "memoize" their solutions in a table. Our algorithm might now look like the following:

DP-FindOpt

Input: A set of requests with weights, $R = \{(r_1, w_1) \dots, (r_n, w_n)\}$ sorted by increasing finish time; a table M s.t. $M[k]$ is either "empty" or contains (A, W), an optimal set of requests A from $\{(r_1, w_1), \dots, (r_k, w_k)\}$ and their weight, W.

If $R = \phi$, put $M[0] \leftarrow (\phi, 0)$ and return M
If $M[n-1]$ = empty, put $M \leftarrow$ DP-FindOpt$(R - \{(r_n, w_n)\}, M)$
If $M[p(n)]$ = empty, put $M \leftarrow$ DP-FindOpt$(\{(r_1, w_1), \dots, (r_{p(n)}, w_{p(n)})\}, M)$
If $M[n-1]_W > M[p(n)] + w_n$, then put $M[n] \leftarrow M[n-1]$
Else Put $M[n] \leftarrow (M[p(n)]_A \cup \{r_n\}, M[p(n)]_W + w_n)$
Return M

Observe that DP-FindOpt only makes a recursive call on the prefix of the first k requests at most once, since M is passed through all of the recursive calls, $M[k]$ is set on this call, and once $M[k]$ is not empty, it is never called again. Thus, DP-FindOpt is invoked at most n times. Since we can sort the requests in time $O(n \log n)$, determine $p(n)$ in time $O(\log n)$, and update M in time $O(n)$, overall, this algorithm can be implemented in time $O(n^2)$.

And, why is it correct? For the same reason as the recursive algorithm: by induction on the size of R, we assume DP-FindOpt fills in the table with optimal solutions for all sets of fewer than n requests; now either there is an

optimal solution that only needs to use the first n-1 requests— contained in $M[n-1]$, by IH, or else the optimal solution needs to use r_n. In this latter case, we know every request after $p(n)$ conflicts with r_n, and every set of the first $p(n)$ requests can be extended by r_n — so any optimal solution is an optimal solution to $\{(r_1,w_1),\dots,(r_{p(n)},w_{p(n)})\}$ extended by r_n. (otherwise, it could be improved.) By IH, $M[p(n)]$ contains such an optimal solution and its weight. Thus, $M[n-1]_W > M[p(n)]_W + w_n$ if and only if we are in the first case (and $M[n]=M[n-1]$ is a solution), and otherwise $M[n]=(M[p(n)]_A \cup \{r_n\}, M[p(n)]_W + w_n)$ is a solution ☒

Taking a step back, we observe that the recursive structure of this algorithm is no longer essential: like the inductive proof of correctness, we could simply start by filling in $M[0]$, then $M[1]$, and so on iteratively, up to $M[n]$.

Iterative-Find Opt:

Input: a set of weighted requests $R=\{(r_1,w_1),\dots,(r_n,w_n)\}$

Initialize $M[0] \leftarrow (\emptyset, 0)$, sort R by increasing finish time.
For $i=1,\dots,n$
  If $M[i-1]_W > M[p(i)]_W + w_i$, Put $M[i] \leftarrow (M[i-1]_A, M[i-1]_W)$
  Else, Put $M[i] \leftarrow (M[p(i)]_A \cup \{r_i\}, M[p(i)]_W + w_i)$
Return $M[n]$.

Its correctness follows the same inductive argument,

and it's immediately clear that it runs in time $O(n^2)$.

Actually, we observe that we can reduce the time to $O(n \log n)$ by making two passes as follows:

Optimal · Find Opt

Input: A set of weighted requests $R = \{(r_1, w_1), \dots, (r_n, w_n)\}$

Initialize $M[0] \leftarrow 0$, sort R by finish time.
For $i = 1, \dots, n$
  Put $M[i] \leftarrow \min\{M[i-1], M[p(i)] + w_i\}$
Initialize $j \leftarrow n$, $A \leftarrow \emptyset$
While $j > 0$
  If $M[j] = M[j-1]$, Put $j \leftarrow j-1$.
  Else put $j \leftarrow p(j)$ and $A \leftarrow A \cup \{r_j\}$
Return A.

By the same argument, M now simply contains the optimal weight using each prefix of R. But, we observe that if $M[i] = M[i-1]$, then the optimal subset of $\{r_1, \dots, r_{i-1}\}$ is of the same weight as any optimal subset of $\{r_1, \dots, r_i\}$, and so does not need to include r_i; by contrast, if not, then any optimal subset of $\{r_1, \dots, r_i\}$ must use r_i, which then has a conflict with precisely the requests following $p(i)$. So now, the optimal set of requests is a subset of $\{1, \dots, p(i)\}$, together with r_i. Precisely now, it follows by induction on j that following an iteration j of the second loop, it adds an optimal set of requests from $\{r_1, \dots, r_j\}$ to A — so, starting from $j = n$, it indeed adds the optimal subset of $\{r_1, \dots, r_n\}$.

While the recursive algorithm helped us to motivate our final algorithm, the final algorithm that iteratively builds up solutions to subproblems is more straightforward. Henceforth, all of the dynamic programming algorithms we describe will be of this form.

In general, we will be able to use dynamic programming when... (1) there are only a polynomial number of subproblems

(2) there is a natural ordering on the subproblems together with an easily computed recurrence giving the solution to each subproblem in terms of smaller subproblems

(3) the solution to the original problem can be easily computed from the solution to the subproblems.