Scheduling Theory **CO 454, Spring 2009**

Lecture 6: Dynamic Programming

May 21st, 2009

---

1 Dynamic Programming

In this lecture we will look at a strong technique in algorithm design called *dynamic programming*. Most books on introductory algorithms (for example, *Introduction to Algorithms* by Cormen-Leiserson-Rivest-Stein or *Algorithm Design* by Kleinberg and Tardos) cover this topic extensively and we suggest referring those books for a much more thorough treatment.

Dynamic programming for optimization problems are applicable when the problem can be *broken down* into many sub-problems and the optimal solution for the original problem *contains within it* optimal solutions of the subproblems. The second condition is called the *optimal sub-structure* and is crucial for any dynamic programming method to work. To solve the given problem, a dynamic program solves smaller subproblems in a bottom-up fashion. The smallest subproblems are most often trivial to solve. Since larger subproblems contain the optimum of smaller subproblems, by storing the older solutions in a *table* the dynamic program can build the optimal solution to the larger subproblem, and so on till the original problem is solved.

We illustrate this technique by showing how it is used to solve the knapsack problem.

Definition 1.1. In the knapsack problem we have a set J of n items. Each item j has a profit p_j and a weight w_j which we assume to be integers. We are also given a knapsack of total capacity B. For this lecture we will assume $B = \text{poly}(n)$. The goal is to choose a subset of items of maximum profit whose total weight is at most B. An instance is denoted as (B, J).

Suppose S is the optimal solution to the knapsack problem (B, J). Suppose J is ordered $\{1, \ldots, n\}$. There are two possibilities for the nth item – either it is in S, or it is not. Moreover, if n is in S, then the remaining solution $S \setminus n$ is also an *optimum* solution to the smaller knapsack problem $(B - w_n, J \setminus n)$; if n is not in S, then S is also an optimum solutio to $(B, J \setminus n)$. Thus the problem satisfies the *optimal substructure property.* Dynamic programming can be used to solve the problem as follows.

A table $T[i, W]$ is maintained with i going from 1 to n and W going from 1 to B. $T[i, W]$ is supposed to contain the maximum profit subset of items $\{1, \ldots, i\}$ with total weight at most W. If we can build this table fully then $T[n, B]$ will give the desired solution. We also let $t[i, W]$ be the profit of $T[i, W]$. Note that $T[0, W]$ is the empty set and $t[0, W] = 0$ for any W. Similarly, $T[i, 0]$ is empty and $t[i, 0] = 0$ for any i. These are the smallest subproblems and the entries of the table we can fill in easily. To fill in the rest of the table, we use the following rule. Suppose all entries $T[i,W]$ from $W = 1$ to B have been evaluated.

$$T[i+1,W] = \begin{cases} T[i,W] & \text{if the item } i+1 \text{ is not in the optimal solution} \\ T[i, W - w_{i+1}] \cup w_{i+1} & \text{if } i+1 \text{ is in the optimal solution} \end{cases} \tag{1}$$

To check if item $(i+1)$ is in the optimal solution or not, we need to check which of the two sets $T[i,W]$ or $T[i,W-w_{i+1}] \cup w_{i+1}$ gives higher profit. We can check this by checking which is bigger among $t[i,W]$ and $t[i,W-w_{i+1}] + p_{i+1}$. Note that this can be checked from the table. Thus, in time $O(nB)$ (the size of the table), the knapsack problem can be solved.

Example 1.2. *Consider the instance with $n = 4$ jobs when the knapsack capacity is $B = 3$. The jobs have (weight,profit) as follows.* $\{(3,1),(2,1),(1,2),(1,2)\}$. *The table $T[i,W]$ gets filled as follows:*

i, w	*0*	*1*	*2*	*3*
0	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$
1	$\varnothing$	$\varnothing$	$\varnothing$	$\{1\}$
2	$\varnothing$	$\varnothing$	$\{2\}$	$\{2\}$
3	$\varnothing$	$\{3\}$	$\{3\}$	$\{2,3\}$
4	$\varnothing$	$\{3\}$	$\{3,4\}$	$\{3,4\}$

Exercise 1.3. *Solve the knapsack problem with $n = 4$ and $B = 11$, with (weights,profits) as follows:* $(3,2),(4,3),(5,4),(7,5)$. *Note that you can pre-fill most of the table (that is solve smaller subproblems) easily. The final answer should have profit* 8.

The main crux in designing a dynamic program is to decide how the table should be constructed (that is, how the subproblems should be defined). In the above problem, the table was relatively straightforward. Sometimes however, the table is a little subtle. We see one such in the next lecture when we show how dynamic programming can be used to solve $(1||\sum w_j U_j)$.

As we saw, optimal substructure was a crucial ingredient for the above dynamic program. Not all problems have an optimal substructure. As an example we look at the following variant of the knapsack problem, called (by me) the colourful knapsack problem.

Definition 1.4. The input is as in the knapsack problem, except each item now also has a color and the goal is to choose a subset of items with total weight at most B and no two items of the same colour are chosen.

The colourful knapsack problem doesn't have the optimal substructure as the knapsack problem. To see an example, consider four items $\{1,2,3,4\}$ each of weight 1. The profits are $1.5, 1, 1, 2$ respectively, and the colors are *red, blue, green, red*. Suppose $B = 3$. Now the optimum solution to $(\{1,2,3,4\}, B = 3)$ is the set $\{2,3,4\}$. However, if we look at the subproblem $(\{1,2,3\}, B = 2)$ which arises when we need to decide of the item $\{4\}$ is in the optimum or not, we get that the optimum of $(\{1,2,3\}, B = 2)$ is now $\{1,2\}$ and is not contained in $\{2,3,4\}$. Thus the same DP cannot be used to solve the colourful knapsack problem.

We end this lecture with a note of caution: one should take the existence of optimal substructure only as an evidence that there *could* be a DP. Similarly, the lack of an optimal substructure, only implies one needs to understand the optimal solution much more, rather than excluding the possibility of a DP. For instance, in the above problem, maybe one could group all the similar colour items together and create the subproblems in a new fashion. Does it work?

2 Minimizing the Weight of Late Jobs ($1|d_j = D|\sum w_jU_j$)

Recall that in a schedule a job j is late ($U_j = 1$) if $C_j > d_j$. We wish to find a schedule which minimizes the total weight of the late jobs. In the next lecture, we will show a DP to solve the problem $1||\sum w_jU_j$. Now, we look at a special case when all the due-dates, d_j are the same D for all jobs and observe that the problem is then equivalent to knapsack.

Note that minimizing the total weight of late jobs is *equivalent* to maximizing the total weight of timely jobs. A job is timely iff its completion time is less than D. A set T of jobs can be timely if and only if the last job completes by time D, in other words, the total processing time of jobs in T is at most D. Thus, the problem of maximizing the total weight of timely jobs is the knapsack problem with knapsack capacity D and each item corresponds to a job, the knapsack-weight of the item is the processing time of the job, and the knapsack-profit of the item is the weight of the job. Thus, the problem can be solved in $O(nD)$ time.