CS 6901 (Applied Algorithms) – Lecture 11

Antonina Kolokolova[*]

October 23, 2014

1 Dynamic programming algorithms

The setting is as follows. We wish to find a solution to a given problem which optimizes some quantity Q of interest; for example, we might wish to maximize profit or minimize cost. The algorithm works by *generalizing* the original problem. More specifically, it works by creating an array of related but simpler problems, and then finding the optimal *value* of Q for each of these problems; we calculate the values for the more complicated problems by using the values already calculated for the easier problems. When we are done, the optimal value of Q for the original problem can be easily computed from one or more values in the array. We then use the array of values computed in order to compute a *solution* for the original problem that attains this optimal value for Q. We will always present a dynamic programming algorithm in the following 4 steps.

Step 1:
Describe an array (or arrays) of values that you want to compute. (Do not say how to compute them, but rather describe what it is that you want to compute.) Say how to use certain elements of this array to compute the optimal value for the original problem.

Step 2:
Give a recurrence relating some values in the array to other values in the array; for the simplest entries, the recurrence should say how to compute their values from scratch. Then (unless the recurrence is obviously true) justify or prove that the recurrence is correct.

Step 3:
Give a high-level program for computing the values of the array, using the above recurrence. Note that one computes these values in a bottom-up fashion, using values that have already been computed in order to compute new values. (One does not compute the values

[*]This set of notes is based on the course notes of U. of Toronto CS 364 as taught by Stephen Cook

recursively, since this would usually cause many values to be computed over and over again, yielding a very inefficient algorithm.) Usually this step is very easy to do, using the recurrence from Step 2. Sometimes one will also compute the values for an auxiliary array, in order to make the computation of a solution in Step 4 more efficient.

Step 4:
Show how to use the values in the array(s) (computed in Step 3) to compute an optimal solution to the original problem. Usually one will use the recurrence from Step 2 to do this.

Moving on a grid example

The following is a very simple, although somewhat artificial, example of a problem easily solvable by a dynamic programming algorithm.

Imagine a climber trying to climb on top of a wall. A wall is constructed out of square blocks of equal size, each of which provides one handhold. Some handholds are more dangerous/complicated than other. From each block the climber can reach three blocks of the row righ above: one right on top, one to the right and one to the left (unless right or left are no available because that is the end of the wall). The goal is to find the least dangerous path from the bottom of the wall to the top, where danger rating (cost) of a path is the sum of danger ratings (costs) of blocks used on that path.

We represent this problem as follows. The input is an $n \times m$ grid, in which each cell has a positive cost $C(i,j)$ associated with it. The bottom row is row 1, the top row is row n. From a cell (i,j) in one step you can reach cells $(i+1, j-1)$ (if $j > 1$), $(i+1, j)$ and $(i+1, j+1)$ (if $j < m$).

Here is an example of an input grid. The easiest path is highlighted. The total cost of the easiest path is 12. Note that a greedy approach – choosing the lowest cost cell at every step – would not yield an optimal solution: if we start from cell $(1,2)$ with cost 2, and choose a cell with minimum cost at every step, we can at the very best get a path with total cost 13.

Grid example.

2	8	9	**5**	8
4	4	6	**2**	3
5	7	5	6	**1**
3	2	5	**4**	8

Step 1. The first step in designing a dynamic programming algorithm is defining an array to hold intermediate values. For $1 \le i \le n$ and $1 \le j \le m$, define $A(i,j)$ to be the cost of the cheapest (least dangerous) path from the bottom to the cell (i,j). To find the value of the best path to the top, we need to find the minimal value in the last row of the array, that is, $\min_{1 \le j \le m} A(n,j)$.

Step 2. This is the core of the solution. We start with the initialization. The simplest way is to set $A(1,j) = C(1,j)$ for $1 \le j \le m$. A somewhat more elegant way is to make an additional zero row, and set $A(0,j) = 0$ for $1 \le j \le m$.
There are three cases to the recurrence: a cell might be in the middle (horizontally), on the leftmost or on the rightmost sides of the grid. Therefore, we compute $A(i,j)$ for $1 \le i \le n$, $1 \le j \le m$ as follows:

$A(i,j)$ for the above grid.

∞	0	0	0	0	0	∞
∞	3	2	5	**4**	8	∞
∞	7	9	7	10	**5**	∞
∞	11	11	13	**7**	8	∞
∞	13	19	16	**12**	15	∞

$$A(i,j) = \begin{cases} C(i,j) + \min\{A(i-1,j-1), A(i-1,j)\} & \text{if } j = m \\ C(i,j) + \min\{A(i-1,j), A(i-1,j+1)\} & \text{if } j = 1 \\ C(i,j) + \min\{A(i-1,j-1), A(i-1,j), A(i-1,j+1)\} & \text{if } j \ne 1 \text{ and } j \ne m \end{cases}$$

We can eliminate the cases if we use some extra storage. Add two columns 0 and $m+1$ and initialize them to some very large number ∞; that is, for all $0 \le i \le n$ set $A(i,0) = A(i,m+1) = \infty$. Then the recurrence becomes, for $1 \le i \le n$, $1 \le j \le m$,

$$A(i,j) = C(i,j) + \min\{A(i-1,j-1), A(i-1,j), A(i-1,j+1)\}$$

Step 3 . Now we need to write a program to compute the array; call the array B. Let INF denote some very large number, so that $INF > c$ for any c occurring in the program (for example, make INF the sum of all costs +1).

```
// initialization
for $j = 1$ to $m$ do
    $B(0,j) \leftarrow 0$
for $i = 0$ to $n$ do
    $B(i,0) \leftarrow INF$
    $B(i,m+1) \leftarrow INF$
// recurrence
for $i = 1$ to $n$ do
    for $j = 1$ to $m$ do
        $B(i,j) \leftarrow C(i,j) + \min\{B(i-1,j-1), B(i-1,j), B(i-1,j+1)\}$
// finding the cost of the least dangerous path
$cost \leftarrow INF$
for $j = 1$ to $m$ do
    if ($B(n,j) < cost$) then
        $cost \leftarrow B(n,j)$
return $cost$
```

Step 4. The last step is to compute the actual path with the smallest cost. The idea is to retrace the decisions made when computing the array. To print the cells in the correct order, we make the program recursive. Skipping finding j such that $A(n,j) = cost$, the first call to the program will be *PrintOpt*(n, j).

```
procedure PrintOpt(i,j)
    if $(i = 0)$ then return
    else if $(B(i,j) = C(i,j) + B(i-1, j-1))$ then PrintOpt(i-1,j-1)
    else if $(B(i,j) = C(i,j) + B(i-1, j))$ then PrintOpt(i-1,j)
    else if $(B(i,j) = C(i,j) + B(i-1, j+1))$ then PrintOpt(i-1,j+1)
    end if
    put "Cell " $(i,j)$
end PrintOpt
```

Scheduling Jobs With Deadlines, Profits, and Durations

In the notes on Greedy Algorithms, we saw an efficient greedy algorithm for the problem of scheduling unit-length jobs which have deadlines and profits. We will now consider a generalization of this problem, where instead of being unit-length, each job now has a *duration* (or processing time).

More specifically, the input will consist of information about n jobs, where for job i we have a nonnegative real valued profit $g_i \in \mathbb{R}^{\geq 0}$, a deadline $d_i \in \mathbb{N}$, and a duration $t_i \in \mathbb{N}$. It is convenient to think of a schedule as being a sequence $C = C(1), C(2), \cdots, C(n)$; if $C(i) = -1$, this means that job i is not scheduled, otherwise $C(i) \in \mathbb{N}$ is the time at which job i is scheduled to begin.

We say that schedule C is *feasible* if the following two properties hold.

(a) Each job finishes by its deadline. That is, for every i, if $C(i) \geq 0$, then $C(i) + t_i \leq d_i$.

(b) No two jobs overlap in the schedule. That is, If $C(i) \geq 0$ and $C(j) \geq 0$ and $i \neq j$, then either $C(i)+t_i \leq C(j)$ or $C(j)+t_j \leq C(i)$. (Note that we permit one job to finish at exactly the same time as another begins.)

We define the profit of a feasible schedule C by $P(C) = \sum_{C(i)\geq 0} g_i$.

We now define the problem of Job Scheduling with Deadlines, Profits and Durations:

Input A list of jobs (d_1, t_1, g_1),..., (d_n, t_n, g_n)

Output A feasible schedule $C = C(1), ..., C(n)$ such that the profit $P(C)$ is the maximum possible among all feasible schedules.

Before beginning the main part of our dynamic programming algorithm, we will sort the jobs according to deadline, so that $d_1 \leq d_2 \leq \cdots \leq d_n = d$, where d is the largest deadline. Looking ahead to how our dynamic programming algorithm will work, it turns out that it is important that we prove the following lemma.

Lemma 1 *Let C be a feasible schedule such that at least one job is scheduled; let $i > 0$ be the largest job number that is scheduled in C. Say that every job that is scheduled in C finishes by time t. Then there is feasible schedule C' that schedules exactly the same jobs as C, and such that $C'(i) = \min\{t, d_i\} - t_i$, and such that all other jobs scheduled by C' end at or before time $\min\{t, d_i\} - t_i$.*

Proof: This proof uses the fact the the jobs are sorted according to deadline. The details are left as an exercise.

We now perform the four steps of a dynamic programming algorithm.

Step 1: Describe an array of values we want to compute.
Define the array $A(i, t)$ for $0 \leq i \leq n$, $0 \leq t \leq d$ by

$$A(i,t) = \max\{P(C) | C \text{ is a feasible schedule in which only jobs from } \{1, \cdots, i\} \text{ are scheduled, and all scheduled jobs finish by time } t\ \}.$$

Note that the value of the profit of the optimal schedule that we are ultimately interested in, is exactly $A(n, d)$.

Step 2: Give a recurrence.
This recurrence will allow us to compute the values of A one row at a time, where by the ith row of A we mean $A(i, 0), \cdots, A(i, d)$.

- $A(0, t) = 0$ for all t, $0 \leq t \leq d$.
- Let $1 \leq i \leq n$, $0 \leq t \leq d$. Define $t' = \min\{t, d_i\} - t_i$. Clearly t' is the latest possible time that we can schedule job i, so that it ends both by its deadline and by time t. Then we have:

If $t' < 0$, then $A(i, t) = A(i - 1, t)$.
If $t' \geq 0$, then $A(i, t) = \max\{A(i - 1, t), g_i + A(i - 1, t')\}$.

We now must explain (or prove) why this recurrence is true. Clearly $A(0, t) = 0$.

To see why the second part of the recurrence is true, first consider the case where $t' < 0$. Then we cannot (feasibly) schedule job i so as to end by time t, so clearly $A(i, t) = A(i-1, t)$. Now assume that $t' \geq 0$. We have a choice of whether or not to schedule job i. If we don't schedule job i, then the best profit we can get (from scheduling some jobs from $\{1, \cdots i\}$ so that all end by time t) is $A(i - 1, t)$. If we do schedule job i, then the previous lemma tells us that we can assume job i is scheduled at time t' and all the other scheduled jobs end by time t', and so the best profit we can get is $g_i + A(i - 1, t')$.

Step 3: Give a high-level program.
We now give a high-level program that computes values into an array B, so that we will have

$B[i,t] = A(i,t)$. (The reason we call our array B instead of A, is to make it convenient to prove that the values computed into B actually are the values of the array A defined above. This proof is usually a simple induction proof that uses the above recurrence, and so usually this proof is omitted.)

```
for every $t \in \{0, \cdots, d\}$
    $B[0,t] \leftarrow 0$
end for
for $i : 1..n$
    for every $t \in \{0, \cdots, d\}$
        $t' \leftarrow \min\{t, d_i\} - t_i$
        if $t' < 0$ then
            $B[i,t] \leftarrow B[i-1,t]$
        else
            $B[i,t] \leftarrow \max\{B[i-1,t], g_i + B[i-1,t']\}$
        end if
    end for
end for
```

Step 4: Compute an optimal solution.
Let us assume we have correctly computed the values of the array A into the array B. It is now convenient to define a "helping" procedure PrintOpt(i,t) that will call itself recursively. Whenever we use a helping procedure, it is important that we specify the appropriate precondition/postcondition for it. In this case, we have:
Precondition: i and t are integers, $0 \leq i \leq n$ and $0 \leq t \leq d$.
Postcondition: A schedule is printed out that is an optimal way of scheduling only jobs from $\{1, \cdots, i\}$ so that all jobs end by time t.

We can now print out an optimal schedule by calling
PrintOpt(n,d)
Note that we have written a recursive program since a simple iterative version would print out the schedule in reverse order. It is easy to prove that PrintOpt(i,t) works, by induction on i. The full program (assuming we have already computed the correct values into B) is as follows:

```
procedure PrintOpt$(i,t)$
    if $i = 0$ then return end if
    if $B[i,t] = B[i-1,t]$ then
        PrintOpt$(i-1,t)$
    else
        $t' \leftarrow \min\{t, d_i\} - t_i$
        PrintOpt$(i-1,t')$
        put "Schedule job", $i$, "at time", $t'$
    end if
end PrintOpt

PrintOpt($n, d$)
```

Analysis of the Running Time
The initial sorting can be done in time $O(n \log n)$. The program in Step 3 clearly takes time $O(nd)$. Therefore we can compute the entire array A in total time $O(nd + n \log n)$. When d is large, this expression is dominated by the term nd. It would be nice if we could state a running time of simply $O(nd)$. Here is one way to do this. When $d \leq n$, instead of using an $n \log n$ sorting algorithm, we can so something faster by noting that we are sorting n numbers from the range 0 to n; this can easily be done (using only $O(n)$ extra storage) in time $O(n)$. Therefore, we can compute the entire array A within total time $O(nd)$. Step 4 runs in time $O(n)$. So our total time to compute an optimal schedule is in $O(nd)$. Keep in mind that we are assuming that each arithmetic operation can be done in constant time.

Should this be considered a polynomial-time algorithm? If we are guaranteed that on all inputs d will be less than, say, n^2, then the algorithm can be considered a polynomial-time algorithm.

More generally, however, the best way to address this question is to view the input as a sequence of bits rather than integers or real numbers. In this model, let us assume that all numbers are integers represented in binary notation. So if the $3n$ integers are represented with about k bits each, then the actual bit-size of the input is about nk bits. It is not hard to see that each arithmetic operation can be done in time polynomial in the bit-size of the input, but how many operations will the algorithm perform? Since d is a k bit number, d can be as large as 2^k. Since 2^k is not polynomial in nk, the algorithm is *not* a polynomial-time algorithm in this setting.

Now consider a slightly different setting. As before, the profits are expressed as binary integers. The durations and deadlines, however, are expressed in *unary* notation. This means that the integer m is expressed as a string of m ones. Hence, d is now less than the bit-size of the input, and so the algorithm is polynomial-time in this setting.

Actually, in order to decide if this algorithm should be used in a specific application, all you really have to know is that it performs about nd arithmetic operations. You can then compare it with other algorithms you know. In this case, perhaps the only other algorithm you know is the "brute force" algorithm that tries all possible subsets of the jobs, seeing which ones can be (feasibly) scheduled. Using the previous lemma, we can test if a set of jobs can be feasibly scheduled in time $O(n)$, so the brute-force algorithm can be implemented to run in time about $n2^n$. Therefore, if d is much less than 2^n then the dynamic programming algorithm is better; if d is much bigger than 2^n then the brute-force algorithm is better; if d is comparable with 2^n, then probably one has to do some program testing to see which algorithm is better for a particular application.