Lecture 13: Dynamic Programming Part 2
10:00 AM, Feb 23, 2018

Contents

1 Holidays
   1.1 Halloween .......................................................... 1
   1.2 Valentine’s Day ..................................................... 4

2 Dynamic Programming Steps ........................................... 7

Objectives

By the end of these notes, you will know:

- the top-down and bottom-up approaches to dynamic programming

By the end of these notes, you will be able to:

- apply dynamic programming to implement faster solutions to problems with recursive solutions than a naive recursive implementation

1 Holidays

An optimization problem is a problem where the goal is to select a best solution among multiple feasible alternatives. Optimization problems arise in all aspects of our lives. Holidays, in particular, are chock full of optimization problems. In this lecture, we will tackle two common holiday optimization problems using DP.

Recall that DP is applicable to problems whose solutions are 1. recursive and 2. can be broken down into overlapping problems. Likewise, DP is applicable to optimization problems whose solutions are 1. recursive and 2. can be broken down into overlapping optimization problems.

When an optimization problem meets these requirements, it is said to exhibit optimal substructure. When a problem exhibits optimal substructure, an optimal solution can be constructed from optimal solutions to its subproblems.

1.1 Halloween

It’s Halloween and you’re going trick-or-treating! Your parents have dropped you off at the north edge of the neighborhood and will pick you up at the south edge in an hour. The houses in the neighborhood are laid out on a rectangular grid, and you only have time to visit one house in each row as you move south procuring candy. Also, lest you risk insulting any of your friendly neighbors
by skipping over them, the next house you visit must be adjacent to the current one, meaning
directly southwest, south, or southeast.

Because you love candy, yet your time is limited, you really want to optimize your path through
the neighborhood to maximize the amount of candy you can procure. To help you and others meet this
lofty goal, the neighborhood’s data scientists have analyzed data from past years to produce reliable
estimates of how much candy each house is expected to give out this year, as shown below:

4 3 1 5
9 15 2 7
2 5 6 17
11 13 4 8

Given this rectangular grid of numbers, a neighborhood whiz kid calculated the optimal path from
top to bottom, abiding by the above-mentioned constraints. That is, she figured out which path
accrues the most candy. We call this problem—given a rectangular grid of numbers, find an optimal
path through the grid from north to south, subject to the constraints of only visiting one house per
row, and of visiting adjacent houses—Candyland.

What is this optimal path? The answer is not immediately obvious. The most straightforward
strategy for searching for an optimal path is what’s called a greedy algorithm, which starts by
picking one of the most candy-filled houses in the north, and then proceeds to one of the next best
adjacent houses. On the given grid, this greedy strategy would reward us with 5, then 7, then 17,
and then 8 candies, for a total of 37. Is this optimal, or can we do better? Actually, how do we
even know? We could enumerate all possible paths, but there are an exponential number of them!

Path-planning problems like Candyland often exhibit optimal substructure: If we know that an
optimal path from A to D visits B—a neighbor of A—and then eventually visits C as well, then
an optimal path from B to D should also visit C. Because of this property, we can construct an
optimal path from A to D by working backwards, considering extensions of only the optimal paths
from each of A’s neighbors to D. Importantly, we need not consider any paths from A to D that
extend suboptimal paths from any of A’s neighbors to D.

Not all path-planning problems exhibit optimal substructure! For example, suppose you want
to fly from Boston to Los Angeles, and you are searching for the lowest fare. If the route with
the lowest fare from Boston to Los Angeles stops first in Chicago and then in Denver, that does
not mean that the route with the lowest fare from Chicago to Los Angeles also stops in Denver;
perhaps the lowest-fare route from Chicago to Los Angeles is non-stop.

Before formalizing the above intuition (that is, before converting the optimal substructure embedded
in Candyland into a recursive solution), let’s encode this description of the neighborhood in a matrix
A with n rows and m columns. If row 0 represents the north edge, and column 0, the west, then
there is a 3 in cell (0, 1) (i.e., $a_{0, 1} = 3$), a 2 in cell (2, 0) (i.e., $a_{2, 0} = 2$), and so on:

<table>
<thead>
<tr>
<th>i=0</th>
<th>j=0</th>
<th>j=1</th>
<th>j=2</th>
<th>j=3</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i=1</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>i=2</td>
<td>9</td>
<td>15</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>i=3</td>
<td>2</td>
<td>5</td>
<td>6</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>13</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>
We will call this input matrix $A$ the *short-term* values of the houses in the neighborhood. What we are really interested in, however, are the *long-term* values of the houses. That is, we would like to calculate the value of trick-or-treating first at house (i.e., cell) $(i, j)$, and trick-or-treating optimally thereafter. Interestingly, these values will allow us to use a greedy algorithm to infer an optimal trick-or-treating path through the neighborhood.

The long-term value $V(i, j)$ of cell $(i, j)$ can be defined recursively as follows: for all $i \in \{0, \ldots, n-1\}$ and $j \in \{0, \ldots, m-1\}$,

- If $i = n - 1$, then the long-term value of cell $(i, j)$ is equal to its short-term value.
- Otherwise, the long-term value of cell $(i, j)$ is the sum of its short-term value and the maximum long-term value that can be achieved among all the cells that can be reached from cell $(i, j)$ in one-step: i.e., among all the cells that are one step down, one step down and one step to the left (if it exists), or one step down and one step to the right (if it exists).

More formally, we compute the long-term values $V(i, j)$ given the short-term values $a_{ij}$ as follows:

$$V(i, j) = \begin{cases} a_{ij}, & \text{if } i = n - 1 \\ a_{ij} + \max\{V(i + 1, j - 1), V(i + 1, j), V(i + 1, j + 1)\}, & \text{otherwise} \end{cases}$$

The recurrence relation in Equation ?? is, in effect, a recursive solution to this optimization problem. However, coding up this recurrence relation directly would not yield a very efficient solution (on the contrary, its run time would exponential), because it would solve many overlapping subproblems more than once. Not to worry, this recurrence relation lends itself to polynomial-time dynamic programming solutions—both top-down and bottom-up. Moreover, the run time of a DP solution is only $O(nm)$, since there are $nm$ subproblems, none of which are ever solved twice, but all of which must be solved at least once to guarantee optimality.

Here, we explore only the bottom-up DP implementation, which, at a high level, proceeds as follows: begin by initializing the values in the bottom row as per Equation ??; then iteratively compute the values in the row above using the values in the row below and Equation ??.

Let’s walk through this algorithm on our sample grid. To get started, we create a new matrix $B$ of the same dimensions as our input $A$. This new matrix will be used to store the long-term values $V$.

Before we begin filling in this matrix, observe that the value of $V(i, j)$ depends on the values of $V(i + 1, j - 1)$, $V(i + 1, j)$, and $V(i + 1, j + 1)$. Hence, the values in our new matrix $B$ must be computed in *row-major* order from bottom to top, so that row $i + 1$’s values are computed before we attempt to compute row $i$’s values.

Given this ordering, we begin filling in our matrix $B$ with the base case values: i.e., the row $i = 3$. By definition, $b_{3,j} = a_{3,j}$ for all $j$. Next, we move on to row $i = 2$. The value in cell $(2, 1)$, for example, is computed as the maximum value of $a_{3,0} = 11$, $a_{3,1} = 13$, and $a_{3,2} = 4$ (i.e., 13), plus the value of $a_{2,1}$ (i.e., 5). Hence, $b_{2,1} = 18$. Similarly, the value in cell $(2, 2)$ is computed as the maximum value of $a_{3,1} = 13$, $a_{3,2} = 4$, and $a_{3,3} = 8$ (i.e., 13), plus the value of $a_{2,2}$ (i.e., 6). Hence, $b_{2,2} = 19$. And so on:
That is all good and well: the long-term value of, say, cell (1, 1) equals 34. But what does that tell us about the optimal trick-or-treating path? Well, the optimal path is in fact encoded in the matrix of long-term values. We just have to extract it.

The first key observation necessary to this extraction process is: your parents should drop you off at cell (0, 0), because it is from this cell that you can accrue the greatest long-term value. Now, from cell (0, 0), which path should you take to accrue the promised 38 pieces of candy? The way to answer this question is to figure out which neighbor contributed to the long-term value of cell (0, 0); naturally, it is the one with the highest value. The cells that are reachable from (0, 0) (i.e., it’s neighbors) are (1, 0) and (1, 1); since \( V(1, 1) > V(1, 0) \), the cell on the optimal path to (0, 0) is (1, 0). Next we ask, which cells are reachable from (1, 0), and find that among cells (2, 0), (2, 1), and (2, 2), \( V(2, 2) \) is greatest; hence, (2, 2) is on the optimal path. To compute the optimal path, we proceed in this way until \( i = n - 2 \) (inclusive), at which point we have made our final decision. (There is no decision to be made when \( i = n - 1 \).)

At long last, we have discovered that the optimal trick-or-treating path through your neighborhood is: (0, 0), (1, 1), (2, 2), (3, 1).

1.2 Valentine’s Day

Suppose you’d like to compile a CD \(^1\) full of songs for someone special on Valentine’s day. You have a huge collection of songs of varying sizes to choose from, but a CD of only finite capacity. Fortunately, you have ratings for each song in the collection, so you know how much “value” your special someone will get from each. Which songs should you include in your CD?

Alternate story: You’ve just received word that the CIT is on fire. On your desk, you have a number of books (CLR, the GNU Emacs Manual, Practical UNIX Programming, etc.), each of which has a value and a weight. The problem is that you only have one backpack with you, and it can only hold only so much. Which books should you rescue?

The knapsack problem (KP) is defined as follows: “Given a knapsack of fixed capacity, \( W \), and a set of \( n \) items, each with a weight and a value, what items should you pack in your knapsack so as to maximize the total value of the items taken?” This version of the problem is technically the 0-1 KP (hereafter KP), because items are indivisible; you either take ’em or leave ’em\(^2\).

The naïve, brute-force solution to a KP is to enumerate all subsets of items, compute the total weight and total value of each subset, and then identify the one with the maximum total value and that can fit in the knapsack. It is easy to see that this approach is exponential: \( O(2^n) \), where \( n \) is the number of items, since \( 2^n \) is the number of subsets of a set of \( n \) items.

---

\(^1\)Remember those?

\(^2\)If items are divisible, so that you can take partial items, the problem is called the continuous KP. This problem is in P; meaning it can be solved in polynomial time. The 0-1 KP is in NP, meaning it is thought (though not proven) to not be solvable in polynomial time.
You might be thinking, why not just grab the most valuable item, and the next most valuable, and so on. Or why can’t you just look at the ratio of value to weight, and take the items with the best ratios? Nice try! These are reasonable approaches but unfortunately, they are not necessarily optimal.

It is possible to set up weights and values such that any heuristic like this would fail. For example, if you have a knapsack whose capacity is 10 pounds, and three items with the following weight-value pairs: (8, 9.9), (6, 6), and (4, 4), then both heuristics mentioned above would include only the first item in the knapsack, whereas the (unique) optimal solution is to take the second and third.

This type of an algorithm is called a greedy algorithm. The main idea is to iterate through the items in some order, choosing one that is best at each iteration, until the knapsack is full. Greedy algorithms are fast, and so they are widely used in practice, in spite of their suboptimality in general.

Another approach to KP, rather than either a naïve, brute-force search or a greedy heuristic, is a recursive solution. Let \( KP(k, w) \) denote an instance of the knapsack problem with \( k \in \{1, \ldots, n\} \) items and capacity \( w \in \{0, \ldots, W\} \). Like CandyLand, \( KP(n, W) \) exhibits optimal substructure.

If \( X \) is an optimal solution to \( KP(k, w) \), either \( X \) does not include \( k \), in which case \( X \) contains an optimal solution to \( KP(k-1, w) \); or, \( X \) does include \( k \), in which case \( X \) contains an optimal solution to \( KP(k-1, w-w_k) \). Given this substructure, we can construct solutions to \( KP(k, w) \) from solutions to \( KP(k-1, w) \) and \( KP(k-1, w-w_k) \). Here is the recurrence relation that exploits this substructure to compute the value of an optimal solution to \( KP(k, w) \):

- The optimal value of \( KP(1, w) \) is either:
  - If \( w_1 > w \), then the only possible solution is the empty set, with value 0;
  - If \( w_1 \leq w \), then the optimal solution is to take item 1, with value \( v_1 \).
- For \( k \in \{2, \ldots, n\} \), the optimal value of \( KP(k, w) \) is given by the better of the following two options, one of which takes item \( k \), and the other of which does not:
  - Don’t take item \( k \): along this branch, the optimal value is the value of \( KP(k-1, w) \);
  - Take item \( k \): along this branch, the optimal value is \( v_k \) plus the value of \( KP(k-1, w-w_k) \).

(We assume the value of \( KP(k, w) = -\infty \) whenever \( w < 0 \).)

More formally, for all \( k \in \{1, \ldots, n\} \) and \( w \in \{0, \ldots, W\} \), we define the matrix \( U(k, w) \) as follows:

\[
U(k, w) = \begin{cases} 
-\infty, & \text{if } w < 0 \\
0, & \text{if } w = 0 \text{ or } k = 0 \\
\max\{U(k-1, w), U(k-1, w-w_k) + v_k\}, & \text{otherwise}
\end{cases}
\]  

(2)

The matrix \( U \) stores the optimal values of subproblems of \( KP(n, W) \). Hence, the optimal value of \( KP(n, W) \) resides in cell \((n, W)\) of this matrix. But what items were taken to achieve this optimal value? To reconstruct the items taken in the solution from the \( U \) values, we will make use of the following observation: \( U(k, w) > U(k-1, w) \) iff item \( k \) was added to the knapsack to increase the solution’s value. Thus we can walk back through the \( U \) matrix, starting from cell \((n, W)\), listing the items \( k \) for which \( U(k, w) > U(k-1, w) \).

\footnote{Suppose not: i.e., suppose \( X \setminus \{k\} \) is \textit{not} an optimal solution to \( KP(k-1, w-w_k) \). Then there exists some other subset \( X' \) with value \( v_{X'} > v_X \), and weight \( w_{X'} \) no more than capacity \( w - w_k \). By adding \( k \) to \( X' \) instead of \( X \), we obtain value \( v_{X' + k} > v_{X + k} \), and \( w_{X' + k} \leq w - w_k + w_k = w \). This is a contradiction, because \( X \) is optimal.}
**Example**  Suppose we have a knapsack with capacity $W = 3$ and the following three items:

$I_1 : (v_1 = 5, w_1 = 3)$  
$I_2 : (v_2 = 8, w_2 = 1)$  
$I_3 : (v_3 = 13, w_3 = 2)$

To solve this problem instance, we will build a $3 \times 3$ matrix, $U$, with rows indexed by items and columns by capacities. The value at cell $(i, j)$ will be the highest achievable value assuming capacity at most $j$ using a subset of the first $i$ items.

Here is the $U$ matrix with the base cases filled in:

<table>
<thead>
<tr>
<th></th>
<th>$w = 0$</th>
<th>$w = 1$</th>
<th>$w = 2$</th>
<th>$w = 3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$k = 0$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$k = 1$</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$k = 2$</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$k = 3$</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Intuitively, row 1 is easy to fill in, because the item 1 $(v_1 = 5, w_1 = 3)$ only fits in the knapsack when its capacity is 3:

<table>
<thead>
<tr>
<th></th>
<th>$w = 0$</th>
<th>$w = 1$</th>
<th>$w = 2$</th>
<th>$w = 3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$k = 0$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$k = 1$</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$k = 2$</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$k = 3$</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

So let’s focus on row 2. That is, let’s consider taking the second item $(v_2 = 8, w_2 = 1)$. By Eqn ??,

$U(2, 1) = \max\{U(1, 1), U(1, (1 - 1)) + 8\} = \max\{0, 0 + 8\} = 8$

Continuing with the rest of the row, $U(2, 2)$ is of value 8 as well:

$U(2, 2) = \max\{U(1, 2), U(1, (2 - 1)) + 8\} = \max\{0, 0 + 8\} = 8$

And when we compute $U(2, 3)$, again we get 8:

$U(2, 3) = \max\{U(1, 3), U(1, (3 - 1)) + 8\} = \max\{5, 0 + 8\} = 8$

Filling in the $U$ matrix, we now have:

<table>
<thead>
<tr>
<th></th>
<th>$w = 0$</th>
<th>$w = 1$</th>
<th>$w = 2$</th>
<th>$w = 3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$k = 0$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$k = 1$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>$k = 2$</td>
<td>0</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>$k = 3$</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

We continue filling in the $U$ matrix in row-major order according to Equation ?? to finally arrive at:
After filling in such a table, the value of the optimal solution to an instance of KP lies in the lower right-hand corner of the $U$ matrix—in this case, $U(3, 3) = 21$. To recover the items included in that solution, we observe that $U(3, 3) > U(2, 3)$, so item 3 was taken. We then go back to $U(2, 3 - w_3) = U(2, 3 - 2) = U(2, 1)$ and repeat. Since $U(2, 1) > U(1, 1)$, item 2 was also taken. We then go back to $U(1, 1 - w_2) = U(1, 1 - 1) = U(1, 0)$. Since $U(1, 0) = U(0, 0)$, item 1 was not taken. Hence, the set of items taken to achieve the value 21 is $\{2, 3\}$.

**Analyzing Knapsack**  After all this work, you might have been led to believe that the viability of a dynamic-programming solution implies polynomial runtime. Let’s take a closer look at the run time of our DP solution to knapsack, and see if this is indeed the case.

In our DP, there is one subproblem for each (item, weight) pair, so given $n$ items and a capacity of $w$, the run time is $O(nw)$. That looks pretty good, actually. But how do $n$ and $w$ relate to the size of the input? Could either of them be hiding additional cost? The input list of items is proportional to size $n$, so nothing is hiding there. But what about capacity?

Capacity, as an input, is a single number, not a list of all possible weights up to that capacity. So it is the value, not the size, of this number that dictates the ensuing number of subproblems. Worse, because of how base 2 (or 10, or any base bigger than 1) numbers work, its value can be exponentially larger than its size. For example, the number 100 can be represented in 7 bits, 1000 in 10 bits, and so on.

So, we cannot conclude that the run time of our DP solution to knapsack is polynomial. Instead, we say that knapsack has a pseudo-polynomial time solution: one that is polynomial in the value of the input, rather than the size. Be watchful for this distinction as you reason about your programs—an exponential slowdown is never a pleasant surprise!

## 2 Dynamic Programming Steps

In summary, let’s outline the steps taken to arrive at a dynamic programming solution to a problem:

1. Write down a recurrence relation that solves the problem.
2. Use the recursive formulation to identify the subproblems.
3. Construct a table in which to store answers to the subproblems.
4. In bottom-up dynamic programming, we then figure out an order in which to fill in the table, based on the dependencies among the subproblems. In recursion, you start with a hard question, which results in an easier question. In bottom-up DP, we reverse the order of the recursion by ordering our entries from simplest (i.e., the base cases) to most complex.
5. Populate the table, without ever reinventing the wheel (that is, without solving any of the subproblems twice). Then based on the answers in the table, output the problem’s solution.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS18 document by filling out the anonymous feedback form: [http://cs.brown.edu/courses/cs018/feedback](http://cs.brown.edu/courses/cs018/feedback)