CIS 575. Introduction to Algorithm Analysis
Material for March 20, 2024

Dynamic Programming for Binary Knapsack Problem

The topic of this note is given as Exercise 15.2-2 in *Cormen*.

1 Binary Knapsack: Problem

The situation is that we have n items, numbered $1 \dots n$, where each item i has a *weight* called w_i (which must be a positive integer), and a *value* called v_i. Moreover, we are given a *knapsack* with *capacity* W, meaning that it cannot carry items whose total weight exceed W. Our goal is to fill the knapsack such that the total value of its items is as large as possible.

As our running example, let $n = 4$ with the four items given by

i	1	2	3	4
w_i	7	9	5	8
v_i	2	4	1	3

and let $W = 14$. It is not hard to see that the best way to fill the knapsack is to pick items 2 and 3 since then we get a total value of $4 + 1 = 5$ (picking items 1 and 2 would give value 6 but is impossible as their total weight exceeds the capacity; picking items 3 and 4 is possible but the resulting value is only 4).

The problem can be formalized as follows: for each $i \in 1..n$, we must pick an x_i such that either[1] $x_i = 0$, meaning that we don't pick item i, or $x_i = 1$, meaning that we do pick item i; when choosing $x_1 \dots x_n$, we must obey the constraint that the total weight of the items we pick does not exceed W:

$$\sum_{i=1}^{n} x_i \cdot w_i \leq W$$

while trying to **maximize** the total value of the items picked: $\displaystyle\sum_{i=1}^{n} x_i \cdot v_i$

There exists a simple solution to this problem: examine all subsets of $1 \dots n$, consider only those with total weight $\leq W$, and find the one(s) with highest total value. But since there are 2^n subsets of $1 \dots n$, this solution is *exponential* in n and hence not feasible except for rather small n.

We would like to have an algorithm that solves the problem in time *polynomial* in n. But unfortunately, despite prolonged efforts by many very smart computer scientists over the

[1] Since we are considering the *binary* knapsack problem, we only have those two choices; later in the course we shall consider the *fractional* knapsack problem where we may pick say half of an item.

last several decades, no such algorithm has yet been discovered. On the other hand, no one has proved that the problem can *not* be solved in polynomial time. But *if* someone finds a polynomial-time algorithm that solves it *then* a *lot* of other problems will also suddenly be solvable by a known polynomial-time algorithm! (The technical term, to be studied in detail in the graduate version of this course, is that the problem is *NP-hard.*)

2 Binary Knapsack: Solution

Despite the ominous remark at the end of the previous section, we shall still aim at an efficient algorithm for the binary knapsack problem, and for that purpose it appears promising to employ *dynamic programming.*

The 1st step is to find which *sub*problems we need to solve. Quite similar to what we did for "optimal exact change", we shall, for integers i, w with $0 \leq i \leq n$ and $0 \leq w \leq W$, define

$V[i, w]$ is the maximum value possible when:
we can pick from only the items $1 \ldots i$ and the capacity is only w.

Observe that the desired answer can be found as $V[n, W]$. We shall now show how to compute $V[i, w]$.

Some cases are simple: if we don't have any items we cannot get any value; similarly if we don't have any capacity (since each item has positive weight):

$$V[0, w] = 0 \text{ for all } w \in 0 \ldots W \qquad (1)$$
$$V[i, 0] = 0 \text{ for all } i \in 0 \ldots n \qquad (2)$$

Now let us look at the subproblems that may depend on other subproblems; that is, when $i > 0$ and $w > 0$. In that case, we need to examine item i. If that item is too heavy to fit, that is if $w_i > w$, it must be ignored:

$$V[i, w] = V[i-1, w] \text{ when } w_i > w,\ 0 < i \leq n,\ 0 < w \leq W \qquad (3)$$

We are left with the interesting case: when item i will fit. In that case, we may pick it, or we may not pick it... Recall that for "optimal exact change" we had the equation

$$c[i, a] = \mathbf{min}(c[i-1, a], 1 + c[i, a - d_i])$$

which may suggest that for Binary Knapsack we get the equation

$$V[i, w] = \mathbf{min}(V[i-1, w], 1 + V[i, w - w_i])$$

but we need to do several modifications:

1. We want to find the *largest* value (rather than the smallest number of coins) so we must replace "min" by "max":

$$V[i, w] = \mathbf{max}(V[i-1, w], 1 + V[i, w - w_i])$$

2. If we pick item i then we gain value v_i (rather than using one more coin) so we must replace "1" by "v_i":

$$V[i, w] = \mathbf{max}(V[i-1, w], v_i + V[i, w - w_i])$$

3. If we pick item i we cannot pick item i again (whereas we could pick several coins from the same denomination) so we must replace "$V[i$" by "$V[i-1$":

$$V[i,w] = \mathbf{max}(V[i-1,w], v_i + V[i-1, w-w_i]) \tag{4}$$

which is indeed the correct equation.

From equations (1)–(4), we can write a program that tabulates $V[0..n, 0..W]$:

```
for $i \leftarrow 0$ to $n$
   for $w \leftarrow 0$ to $W$
      if $i = 0$ or $w = 0$
         $V[i,w] \leftarrow 0$
      else if $w_i > w$
         $V[i,w] \leftarrow V[i-1,w]$
      else
         $V[i,w] \leftarrow \mathbf{max}(V[i-1,w], v_i + V[i-1,w-w_i])$
```

Note that each entry is indeed defined before it is referenced (we must always check that).
For our example, we get the table below:

$i\backslash w$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	2	2	2	2	2	2	2	2
2	0	0	0	0	0	0	0	2	2	4	4	4	4	4	4
3	0	0	0	0	0	1	1	2	2	4	4	4	4	4	5
4	0	0	0	0	0	1	1	2	3	4	4	4	4	4	5

The bottom right entry is computed as follows:

$$V[4,14] = \mathbf{max}(V[3,14], 3 + V[3,6]) = \mathbf{max}(5, 3+1) = \mathbf{5}$$

which is indeed, as we expected, the optimal value. Still, the table doesn't directly show us *how* to get that optimal value. But we can "retrace our steps" to infer that we should indeed pick items 2 and 3:

1. since $V[4,14] = V[3,14]$, we do *not* need to pick item 4
2. since $V[3,14] > V[2,14]$, we *must* pick item 3, and then examine $V[2, 14-w_3] = V[2,9]$
3. since $V[2,9] > V[1,9]$, we *must* pick item 2, and then examine $V[1, 9-w_2] = V[1,0]$
4. since $V[1,0] = V[0,0]$, we should *not* pick item 1.

Space and Time Use The table has size $(n+1)(W+1)$ which is in $\Theta(n \cdot W)$.

Since each entry can be computed in constant time, we infer that also the running time is in $\mathbf{\Theta(n \cdot W)}$ (and it is easy to see that the solution can be reconstructed in time $\Theta(n)$).

Now you may (should!) wonder: we did recently state that no one had found a solution that runs in polynomial time, but doesn't $n \cdot W$ look polynomial? So have we just made an amazing discovery? Well, unfortunately no... since in principle (unless we make further assumptions), the capacity W could be arbitrarily big! (in particular, not bounded by a polynomial in n).