COMPSCI 330: Design and Analysis of Algorithms **Jan 22, 2020**

Lecture 4: Dynamic Programming I

Lecturer: Rong Ge *Scribe: Mo Zhou*

Overview

In this lecture, we will take Knapsack problem as an example to illustrate the way Dynamic Programming works. First, we will use an instance of Knapsack problem to intuitively show how Dynamic Programming solves this problem. After that, we formalize the algorithm to make it complete. In the end, we will cover the proof of correctness of Dynamic Programming algorithms.

4.1 Knapsack Problem

For Knapsack Problem, there is a knapsack with capacity W, i.e., it can only hold items of total weight at most W. There are n items, whose weights are w_1,w_2, $\cdots$, w_n. Each item also has a value, and the corresponding values are v_1, v_2, $\cdots$, v_n.

Our goal is to select some items to be put into the knapsack such that the total value of those items is maximized. That is, the sum of the weight of those selected items cannot exceed W, and we want the total value of those items to be as large as possible.

4.2 An Example of Knapsack Problem

Now let's see an example of Knapsack problem.

Assume the capacity of the knapsack is 10, i.e., $W = 10$, and there are three items. Item 1 has weight $w_1 = 8$ and value $v_1 = 10$, item 2 has weight $w_2 = 6$ and value $v_2 = 7$, and item 3 has weight $w_3 = 3$ and value $v_3 = 4$. This example is shown in Figure 4.1.

It's easy to see the solution for this example is to select item 2 and 3. If we put item 1 in the knapsack, then there is no enough capacity left for any other item, so the total value is at most $v_1 = 10$. However, if we leave item 1 out of the knapsack, then the optimal strategy is to put both items 2 and 3 into the knapsack. The total weight is

$$w_2 + w_3 = 9 \leq 10,$$

and the total value will be

$$v_2 + v_3 = 11.$$

Knapsack problem is not trivial to solve in a sense that straightforward ways to solve that problem will fail.

One attempt to solve Knapsack problem is to keep selecting the item with the highest value until we have no enough capacity to hold any of the items. However, the above example serves as a counterexample for this method: In that example, we will select item 1 first and then terminate. However, that is not the optimal solution of this problem.

Another attempt will be to keep selecting the item with the highest value-weight ratio. This attempt also fails if we set $v_3 = 3.1$. If we do that, the optimal solution is still selecting items 2 and 3, but our algorithm will select item 1 first and terminate because $\frac{10}{8} > \frac{7}{6} > \frac{3.1}{3}$.

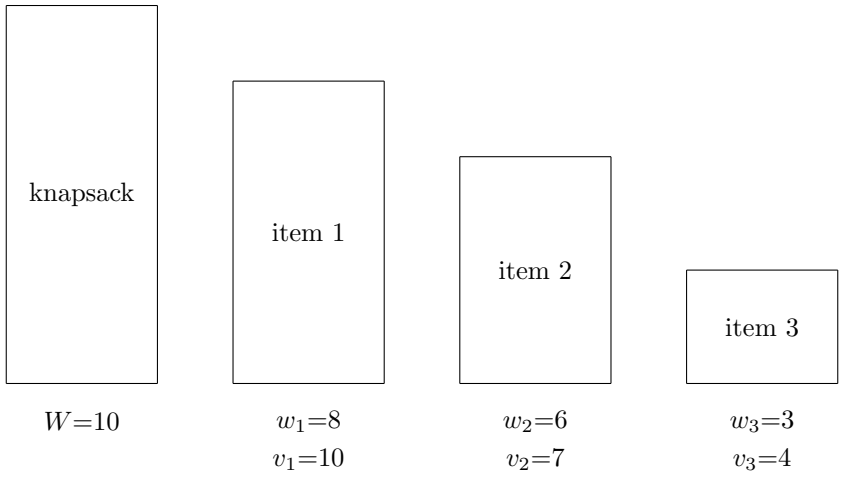

Figure 4.1: Knapsack Problem Example

Thus, Knapsack problem is not easy to solve using straightforward algorithms. Next, we will propose a Dynamic Programming algorithm for Knapsack problem and show how it works.

4.3 Dynamic Programming Algorithm for Knapsack Problem

4.3.1 Steps to Design a Dynamic Programming Algorithm

Step 1: Think of the problem as making a sequence of decisions. In Knapsack problem, for each item, we need to decide whether we put it into the knapsack.
Step 2: Focus on the last decision, and enumerate the options. In this problem, for the last item, we either put it in, or leave it out, so there are two options for each decision.
Step 3: Try to relate each option to a smaller subproblem. For Knapsack problem, the subproblem will be to fill the remaining capacity using the remaining items. If we decide to leave it out, the number of remaining items is smaller. If we decide to put it in, then both the remaining capacity and number of remaining items will be smaller.

4.3.2 Relate Problem to Smaller Subproblems

In this section, we use another example to illustrate how we relate the problem to smaller subproblems. In this example, there are 3 items, with weights $w_1 = 1$, $w_2 = 2$ and $w_3 = 3$, and the corresponding values are $v_1 = 2$, $v_2 = 3$ and $v_3 = 4$. The capacity of our knapsack is $W = 4$. The whole process is shown in Figure 4.2.

As shown in Figure 4.2, the original problem is the leftmost box, where we have capacity 4 and items 1, 2, and 3. At this stage, the last decision is whether we put item 3 into the knapsack, and there are two options: put it in or leave it out. If we put item 3 into the knapsack, then we gain a value of 4, the capacity left is 1 and the remaining items are items 1 and 2. If we leave item 3 out, then we keep all capacity, and the remaining items are also items 1 and 2.

Assume now we have known that if we put item 3 into the knapsack, the maximum value of the related subproblem (i.e., the capacity is 1 and the remaining items are 1 and 2) is 2. Thus, if we put item 3 in, the optimal value will be $2 + 4 = 4 = 6$. Assume we also know that if we leave item 3 out, the optimal value of the related subproblem is 5. Then we can make the decision: Since picking item 3 gives us a better value,

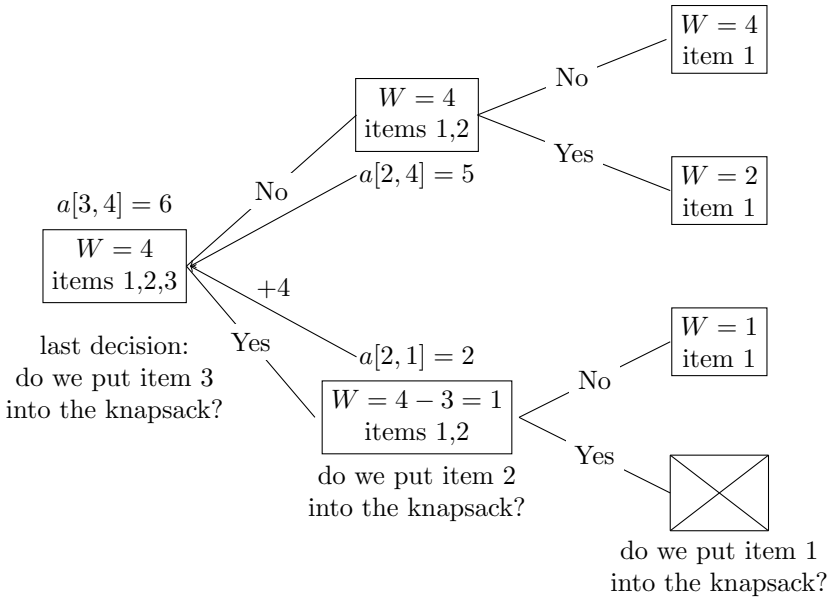

Figure 4.2: State Transition Graph for Knapsack Problem Example

we decide to pick item 3 and the optimal value for the original problem is 6. This is how the recursion tree in Figure 4.2 works.

After we relate the original problem to smaller subproblems, we can further relate those subproblems to much smaller subproblems. This process is the same as above and is drawn in Figure 4.2. For each subproblem, we first consider its last decision, which is whether we put item 2 into the knapsack. Then we enumerate the options, which are yes or no. In the end, we derive the even smaller subproblems from the decisions we make.

Remark: There can be invalid subproblems, e.g., when we have capacity 1 and items 1 and 2 (the mid-bottom box in Figure 4.2) and we decide to put item 2 in, there is no enough space. Also, although in the figure we don't see duplicated subproblems, when the problem becomes more complicated, there will be a lot of duplicated subproblems, which is the reason why Dynamic Programming can speed up this recursion process.

4.3.3 Bound the Number of States

To analyze the running time of this algorithm, we first need to bound the number of states, where states are defined as the subproblems we need to solve.

For each state, the remaining items must be in the form of

$$\{1, 2, \cdots, i\}, \text{ where } i = 0, 1, 2, 3, \cdots, n.$$

Also, the remaining capacity must be in

$$\{0, 1, 2, \cdots, c\}, \text{ where } c = 0, 1, 2, \cdots, W.$$

Thus, the total number of states is at most $O(n) \times O(W) = O(nW)$.

4.4 Formalized Version of the Algorithm

Now, let us formalize the previous algorithm. The formalized version of Dynamic Programming algorithm always consists of four parts: definition of the states, transition function, base cases, and order of computation.

4.4.1 Definition of States

Before describing a Dynamic Programming algorithm, we need to define the states, i.e., the subproblems. In this Knapsack problem, we define $a[i, j]$ to be the maximum possible value of a knapsack with capacity j and can use first i items. Note that the answer to our original problem is $a[n, W]$, where n is the total number of items, and W is the capacity of the knapsack.

4.4.2 Transition Function

The transition function formalizes the way we relates a problem to its subproblems. For this problem, the transition function is

$$a[i,j] \overset{\text{put item } i \text{ into knapsack?}}{=} \max \begin{cases} a[i-1.j] & \text{No} \\ a[i-1, j-w_i] + v_i & \text{Yes (only if } j \geq w_i) \end{cases}. \tag{4.1}$$

Note that we need to check whether $j \geq w_i$, otherwise the subproblem will become invalid.

4.4.3 Base Case

The base cases are the starting points of the Dynamic Programming algorithm. For Knapsack problem, the base cases are either there are no items or there is no capacity, i.e.,

$$\begin{array}{ll} a[0,j] = 0, \forall j \in \{0, 1, \cdots, W\} & \text{(no items).} \\ a[i,0] = 0, \forall i \in \{0, 1, \cdots, n\} & \text{(no capacity).} \end{array}$$

4.4.4 Order of Computation

For Knapsack problem, each state can be related to the states with smaller i indexes or smaller j indexes. Thus, we can compute the value of all states in two for-loops. The pesudocode is listed in Algorithm 1.

Algorithm 1 Dynamic Programming Algorithm for Knapsack Problem

```
for i = 0 to n do
  a[i, 0] = 0
end for
for j = 0 to W do
  a[0, j] = 0
end for
for i = 1 to n do
  for j = 1 to W do
    compute a[i, j] using transition function (4.1).
  end for
end for
return a[n, W]
```

$i \setminus j$	0	1	2	3	4
0	0	0	0	0	0
1	0	2	2	2	2
2	0	2	3	5	5
3	0	2	3	5	6

Table 4.1: Dynamic Programming Table

4.5 Dynamic Programming Table

Dynamic Programming Table is another way of thinking about how Dynamic Programming works. Since each state can be represented by its indexes, the Dynamic Programming algorithms are actually filling in a table cell be cell. Let us still consider the example in section 4.3, where there are 3 items, with weights $w_1 = 1$, $w_2 = 2$ and $w_3 = 3$, and the corresponding values are $v_1 = 2$, $v_2 = 3$ and $v_3 = 4$. The capacity of our knapsack is $W = 4$.

For this Knapsack problem, the corresponding Dynamic Programming table is shown in Table 4.1. First, we fill in the base cases $a[i, 0] = 0$ and $a[0, j] = 0$, which are the first row and the first column of this table. After that, we follow the order of computation and fill in the other cells one by one. In this problem, we are filling row by row from top to bottom, and in each row we fill from left to right.

We will not go over the entire process cell by cell, but we will look at two example cells: $a[1, 1]$ and $a[2, 2]$. Note that

$$a[1,1] = \max \begin{cases} a[0,1] & \text{not using item 1} \\ a[0,0]+2 & \text{using item 1} \end{cases}.$$

$$a[2,2] = \max \begin{cases} a[1,2] & \text{not using item 2} \\ a[1,0]+3 & \text{using item 2} \end{cases}.$$

The arrows in Table 4.1 illustrates the dependencies between states, i.e., when we compute $a[2, 2]$, we must refer back to $a[1, 0]$ and $a[1, 2]$. After we make the decision at a cell, e.g., we find that $a[1, 0] + 3 = 3 > 2 = a[1, 2]$, we store the arrow pointing to $a[1, 0]$. In this way, we can track back from the final cell (in this case, $a[3, 4]$) and output all the decisions we made in order to get to the optimal solution (in this case, all the items we decide to select).

4.6 Proof of Correctness

The general idea to prove a Dynamic Programming algorithm is correct is to use induction in the same order as you compute the states.

Take Knapsack problem as an example, you can refer to Table 4.1 to get some intuition of this proof.

Our Inductive Hypothesis is that we compute all $a[u, v]$s correctly for all pairs of $(u, v) < (i, j)$. Here "$<$" means (u, v) is computed before (i, j).

The base cases for this induction are the base cases for our algorithm, which are $a[i, 0] = a[0, j] = 0$. They are trivially correct.

Assume our Inductive Hypothesis is true, then when computing $a[i, j]$, at this state we either put item i in the knapsack or leave it out. Since we are taking the maximum over the two values, and both $a[i-1, j]$ and $a[i-1, j-w_i]$ are computed correctly based on our Inductive Hypothesis, our algorithm computes the correct value of $a[i, j]$. This finishes the induction.