CS 3510 Algorithms 10/17/2023

Lecture 14: Knapsack

Lecturer: Abrahim Ladha *Scribe(s): Adam Zamlynny*

Knapsack is a core problem to Dynamic Programming, and it's pretty easy to understand. Knapsack is a hard problem though; we don't have or believe there is a polynomial time solution. It's also a good segway into the next unit because there are many problems like it.

1 Problem

Suppose you are given n items (v_i, w_i) where v_i is the value of item i and w_i is the weight of item i. You have a capacity W. Your goal is to maximize the value of items subject to having the sum of the weights be less than or equal to the capacity.

As an example consider if you have $W = 6, w_i = (1, 2, 3, 4), v_i = (12, 17, 18, 25)$. [1] If you picked one and two, you'd have a value of 29. If you picked two and three, you'd have a value of 35 which is better than if you'd picked one and two. If you picked using a greedy strategy, you'd choose four and two which would give you 42. The optimal solution to this is picking one two and three which will give you 47. You could also brute force this by analyzing all 2^n subsets.

2 Solution

Let's start with a two dimensional solution. $T[i, j]$ needs to be an intermediary version of some capacity and some item, so that at the end we could index at $T[W, n]$ giving us the maximum value for the weight and all items. Then let's define our table $T[i, j]$ where $T[i, j]$ is the maximum profit with a capacity i, and only using items $1...j$. In this variant of knapsack, there is only one copy of an item. Then iteratively we can build up the smaller cells in the table to the bigger items.

For our recurrence, let's consider what the last possible step is in the process. Given one more item, if you don't pick the item, the solution is equivalent to the value with the same capacity except for the item ($T[W, n] = T[W, n-1]$). If you do pick the item, your capacity goes up by the value of w_n, so your capacity before this should be $W - w_n$, then your new value is $T[W, n] = T[W - w_n, n-1] + v_n$. This is essentially saying your solution to the optimal value at some weight is the optimal solution with the capacity minus the weight of some new item plus the value of that item, or the weight ignoring that item.

$$T[i, j] = \left\{ \begin{array}{ll} T[i, j-1], & \text{if } w_j > i \\ \max(T[i, j-1], T[i - w_j, j-1] + v_j), & \text{if } w_j \leq i \end{array} \right\}$$

If you notice, in the chain matrix, we have a minimum over some things, but here we have the maximum over some things. Similarly, we're not going to end up finding the items

[1] The weights are just indices here, but that's not always true.

we've used using this table, but like in chain matrix, all it requires is storing pointers to which choices you made. This variant of knapsack is also called 0-1 Knapsack since you can only choose 0 or 1 amounts of any item.

Algorithm 1 Knapsack No Repeat

```
T ← table indexed from 0 to W and 0 to n                ▷ Create the table
while j in 0..n do                                      ▷ Setup base cases T[0, j] = 0
end while
while i in 0..W do T[i, 0] = 0
end while
while j in 1..n do                                      ▷ Fill in the table
    while i in 1..W do
        if w_j ≤ i then
            T[i, j] = max(T[i, j − 1], T[i − w_j, j − 1] + v_j)
        else
            T[i, j] = T[i, j − 1]
        end if
    end while
end while
return T[W, n]
```

The space of this is $\mathcal{O}(nW)$. If you look at this though, you might see that the runtime is dependent on the capacity which is a number. The size of the capacity is dependent on the size of the bits, making the runtime exponential in terms in the bits. W is given as input as k bits, so in terms of the size of the input the complexity is really $O(2^k n)$. Of course this seems like an easy problem, but there is currently no known solution to this problem which takes polynomial time with respect to the size of the input.

3 Example

Suppose we have the weights we had at the beginning. We'd create the following table, and we'd fill in our base cases.

	0	1	2	3	4
0	0	0	0	0	0
1	0				
2	0				
3	0				
4	0				
5	0				
6	0				

	0	1	2	3	4
0	0	0	0	0	0
1	0	12	12	12	12
2	0	12	17	17	17
3	0	12	29	29	29
4	0	12	29	30	30
5	0	12	29	35	37
6	0	12	29	47	47

For the first item, we can't fit it in with capacity zero, but we can fit it in with any capacity more than one so our table becomes the following. Then for item two with capacity two, it can't be fit in with one capacity so 12 comes over. At capacity two, it can be fit so only item two is optimal because it has a greater value; then after capacity three, both are picked. We can then continue filling out the table to get the following. Let's quickly zoom in on two decisions.

	0	1	2	3	4
0	0	[0]	0	0	0
1	0	12	12		
2	0	[12]	[17]		
3	0	12			
4	0	12			
5	0	12			
6	0	12			

	0	1	2	3	4
0	0	0	0	0	0
1	0	12	12	12	12
2	0	12	17	[17]	17
3	0	12	29	29	29
4	0	12	29	30	30
5	0	12	29	35	37
6	0	12	29	[47]	[47]

Figure 1: On the left, picking from item two at capacity two; the algorithm compares $0+17$ and 12 and finds that 17 is larger. On the right, picking from 47 and $17+25$, finding that 47 is larger.

4 Repetition

Now, we allow ourselves to repeat items, so our problem is the same, and our table is the same. The only difference is that our recurrence is now done allowing for repetition, making $j-1$ in the original equation j. The base cases will also be the same with similar reasoning.

$$T[i,j] = \left\{ \begin{array}{ll} T[i,j-1], & \text{if } w_j > i \\ \max(T[i,j-1], T[i-w_j,j]+v_j), & \text{if } w_j \leq i \end{array} \right\}$$

The space of this is $\mathcal{O}(nW)$. The runtime is the same as before. [2]

5 Linear Space

To our knowledge we can't improve runtime, so instead let's try making the space better. Here we'll create one table $T[i]$ which is indexed by the capacity using all the items. Then

[2]Most variants of knapsack are as hard as each other; although one may think there is some greedy approach to this, sadly there is not.

Algorithm 2 Knapsack With Repeat

$T \leftarrow$ table indexed from 0 to W ▷ Create the table
T[0] = 0
while i in 1..W **do**

$$T[i] = \max_{1 \leq j \leq n; w_j \leq i} (T[i - w_j] + v_j)$$

end while
return $T[W, n]$

our recursive statement will be as follows

$$T[i] = \max_{1 \leq j \leq n; w_j \leq i} (T[i - w_j] + v_j)$$

At each step, we consider all the possible elements which could be the last element. Sadly at each step we still need to consider each element, so our runtime is $\mathcal{O}(nW)$. However our space is $\mathcal{O}(W)$.