Algorithm Design-2

(Dynamic Programming (0-1 Knapsack problem))

Presented
By

Dr. Rajesh Purkait
Asst. Professor
Dept. of CSE ITER
(S`O'A Deemed To Be University)
E-mail: rajeshpurkait@soa.ac.in

Outline

Knapsack problem

- Given some items, pack the knapsack to get the **maximum total value.**
- Each item has some weight and some value.
- Total weight that we can carry is no more than some **fixed weight W.**

So we must consider weights of items as well as their values.

Item #	Weight	Value
1	1	8
2	3	6
3	5	5

Knapsack problem

There are two versions of the problem:

1. "0-1 knapsack problem"
   - Items are indivisible; you either take an item or not. Some special instances can be solved with *dynamic programming*

2. "Fractional knapsack problem"
   - Items are divisible: you can take any fraction of an item

0-1 Knapsack problem

- Given a knapsack with maximum capacity W, and a set S consisting of n items
- Each item i has some weight w_i and benefit value b_i (all w_i and W are integer values)
- Problem: How to pack the knapsack to achieve maximum total value of packed items?

0-1 Knapsack problem

- Problem, in other words, is to find

$$\max \sum_{i \in T} b_i \text{ subject to } \sum_{i \in T} w_i \leq W$$

- The problem is called a "*0-1*" problem, because each item must be entirely accepted or rejected.

0-1 Knapsack problem: brute-force approach

Let's first solve this problem with a straightforward algorithm

- Since there are n items, there are 2^n possible combinations of items.
- We go through all combinations and find the one with maximum value and with total weight less or equal to W
- Running time will be $O(2^n)$

0-1 Knapsack problem: dynamic programming approach

- We can do better with an algorithm based on dynamic programming
- We need to carefully identify the subproblems

Defining a Subproblem

- Given a knapsack with maximum capacity W, and a set S consisting of n items
- Each item i has some weight w_i and benefit value b_i (all w_i and W are integer values)

- Problem: How to pack the knapsack to achieve maximum total value of packed items?

Defining a Subproblem

- We can do better with an algorithm based on dynamic programming
- We need to carefully identify the subproblems

Let's try this: If items are labeled *1..n*, then a subproblem would be to find an optimal solution for S_k = *{items labeled 1, 2, .. k}*

Defining a Subproblem

If items are labeled *1..n*, then a subproblem would be to find an optimal solution for S_k = *{items labeled 1, 2, .. k}*

- This is a reasonable subproblem definition.
- The question is: can we describe the final solution (S_n) in terms of subproblems (S_k)?
- Unfortunately, we can't do that.

Defining a Subproblem

$w_1 = 2$ $b_1 = 3$	$w_2 = 4$ $b_2 = 5$	$w_3 = 5$ $b_3 = 8$	$w_4 = 3$ $b_4 = 4$	

?

Max weight: W = 20

For S_4:

Total weight: 14

Maximum benefit: 20

$w_1 = 2$ $b_1 = 3$	$w_2 = 4$ $b_2 = 5$	$w_3 = 5$ $b_3 = 8$	$w_5 = 9$ $b_5 = 10$	

For S_5:

Total weight: 20

Maximum benefit: 26

Item #	Weight w_i	Benefit b_i
1	2	3
2	4	5
3	5	8
4	3	4
5	9	10

S_4

S_5

Solution for S_4 is not part of the solution for S_5!!!

Defining a Subproblem

- As we have seen, the solution for S_4 is not part of the solution for S_5
- So our definition of a subproblem is flawed and we need another one!

Defining a Subproblem

- Given a knapsack with maximum capacity W, and a set S consisting of n items
- Each item i has some weight w_i and benefit value b_i (all w_i and W are integer values)
- Problem: How to pack the knapsack to achieve maximum total value of packed items?

Defining a Subproblem

- Let's add another parameter: *w*, which will represent the maximum weight for each subset of items

- The subproblem then will be to compute *V[k,w], i.e.,* to find an optimal solution for S_k *= {items labeled 1, 2, .. k} in a knapsack of size w*

Recursive Formula for subproblems

- The subproblem will then be to compute *V[k,w], i.e.,* to find an optimal solution for S_k *= {items labeled 1, 2, .. k} in a knapsack of size w*

- Assuming knowing V[i, j], where i=0,1, 2, … k-1, j=0,1,2, …w, how to derive V[k,w]?

Recursive Formula for subproblems (continued)

Recursive formula for subproblems:

$$V[k,w] = \begin{cases} V[k-1,w] & \text{if } w_k > w \\ \max\{V[k-1,w], V[k-1,w-w_k] + b_k\} & \text{else} \end{cases}$$

It means, that the best subset of S_k that has total weight w is:

1) the best subset of S_{k-1} that has total weight $\leq w$, or

2) the best subset of S_{k-1} that has total weight $\leq w - w_k$ plus the item k

Recursive Formula

$$V[k,w] = \begin{cases} V[k-1,w] & \text{if } w_k > w \\ \max\{V[k-1,w], V[k-1,w-w_k] + b_k\} & \text{else} \end{cases}$$

- The best subset of S_k that has the total weight $\leq w$, either contains item k or not.
- **First case:** $w_k > w$. Item k can't be part of the solution, since if it was, the total weight would be $> w$, which is unacceptable.
- **Second case:** $w_k \leq w$. Then the item k can be in the solution, and we choose *the case with greater value*.

0-1 Knapsack Algorithm

```
for w = 0 to W
  V[0,w] = 0
for i = 1 to n
  V[i,0] = 0
for i = 1 to n
  for w = 0 to W
       if w_i <= w // item i can be part of the
  solution
            if b_i + V[i-1,w-w_i] > V[i-1,w]
                 V[i,w] = b_i + V[i-1,w- w_i]
            else
                 V[i,w] = V[i-1,w]
       else V[i,w] = V[i-1,w]  // w_i > w
```

Running time

```
for w = 0 to W
  V[0,w] = 0
for i = 1 to n
  V[i,0] = 0
for i = 1 to n
  for w = 0 to W
      < the rest of the code >
```

O(W)

Repeat *n* times

O(W)

What is the running time of this algorithm?

O(n*W)

Remember that the brute-force algorithm takes **O(2^n)**

Example

Let's run our algorithm on the following data:

n = 4 (# of elements)
W = 5 (max weight)
Elements (weight, benefit):
(2,3), (3,4), (4,5), (5,6)

Comments

- This algorithm only finds the max possible value that can be carried in the knapsack
  - i.e., the value in V[n,W]
- To know the items that make this maximum value, an addition to this algorithm is necessary

How to find actual Knapsack Items

- All of the information we need is in the table.
- $V[n,W]$ is the maximal value of items that can be placed in the Knapsack.
- Let i=n and k=W

  if $V[i,k] \neq V[i-1,k]$ then

    mark the i^{th} item as in the knapsack

    $i = i-1,\ k = k-w_i$

  else

    $i = i-1$ // Assume the i^{th} item is <u>not</u> in the knapsack

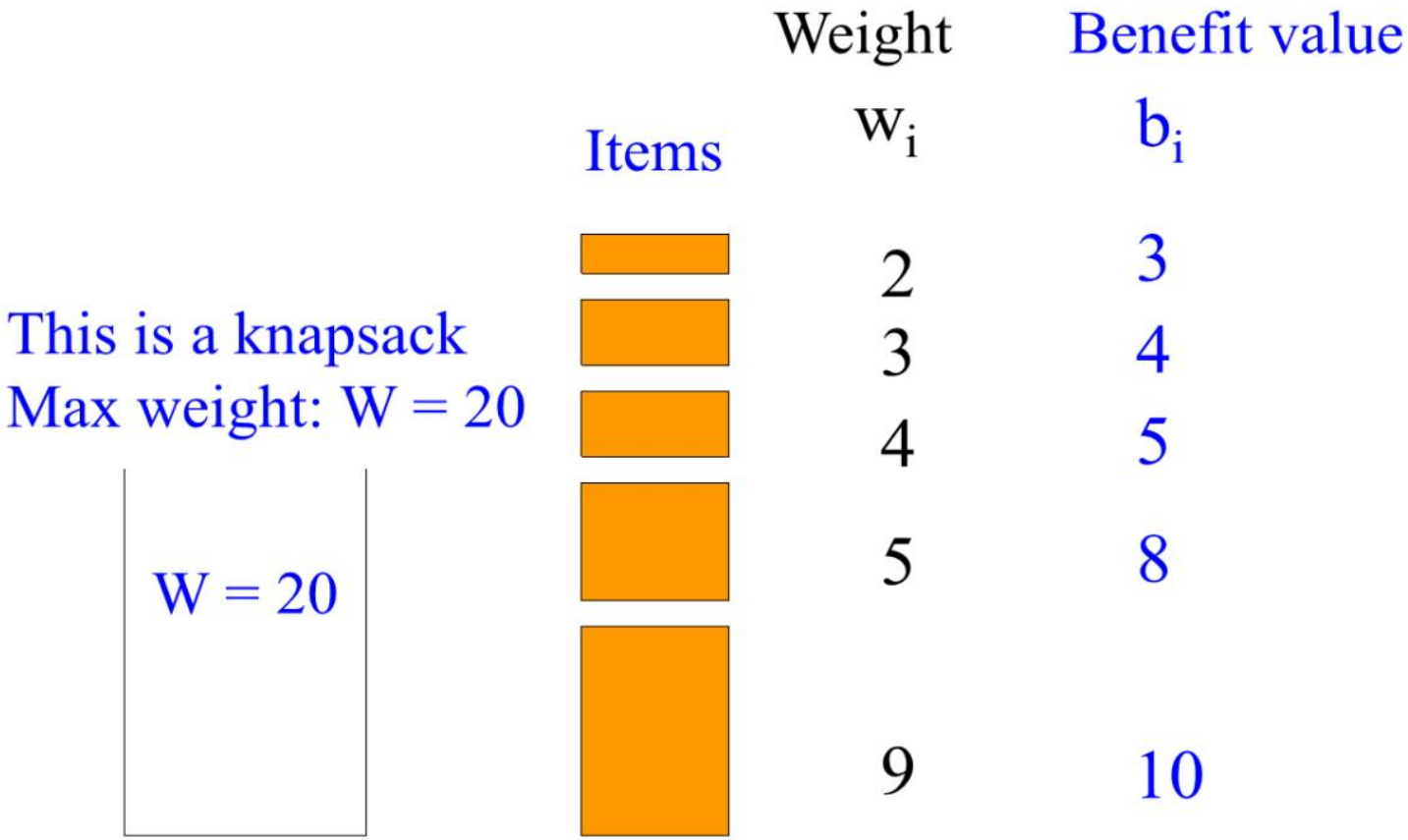
    // Could it be in the optimally packed knapsack?

Conclusion

- Dynamic programming is a useful technique of solving certain kind of problems
- When the solution can be *recursively* described in terms of partial solutions, we can store these partial solutions and re-use them as necessary (memorization)
- Running time of dynamic programming algorithm vs. naïve algorithm:
  - 0-1 Knapsack problem: **O(W*n)** vs. **O(2^n)**

Example 2

0-1 Knapsack problem: a picture



Thank You