

Lecture 13: The Knapsack Problem

Outline of this Lecture

- Introduction of the 0-1 Knapsack Problem.
- A dynamic programming solution to this problem.

0-1 Knapsack Problem

Informal Description: We have computed n data files that we want to store, and we have available W bytes of storage.

File i has size w_i bytes and takes v_i minutes to recompute.

We want to avoid as much recomputing as possible, so we want to find a subset of files to store such that

- The files have combined size at most W .
- The total computing time of the stored files is as large as possible.

We can not store parts of files, it is the whole file or nothing.

How should we select the files?

0-1 Knapsack Problem

Formal description: Given two n -tuples of positive numbers

$$\langle v_1, v_2, \dots, v_n \rangle \quad \text{and} \quad \langle w_1, w_2, \dots, w_n \rangle,$$

and $W > 0$, we wish to determine the subset $T \subseteq \{1, 2, \dots, n\}$ (of files to store) that

$$\text{maximizes} \quad \sum_{i \in T} v_i,$$

$$\text{subject to} \quad \sum_{i \in T} w_i \leq W.$$

Remark: This is an optimization problem.

Brute force: Try all 2^n possible subsets T .

Question: Any solution better than the brute-force?

Recall of the Divide-and-Conquer

1. **Partition** the problem into subproblems.
2. **Solve** the subproblems.
3. **Combine** the solutions to solve the original one.

Remark: If the subproblems are **not independent**, i.e. subproblems share subsubproblems, then a divide-and-conquer algorithm **repeatedly** solves the common subsubproblems.

Thus, it does **more work than necessary!**

Question: Any better solution?

Yes—Dynamic programming (DP)!

The Idea of Dynamic Programming

Dynamic programming is a method for solving optimization problems.

The idea: Compute the solutions to the subsub-problems *once* and store the solutions in a table, so that they can be *reused* (repeatedly) later.

Remark: We trade space for time.

The Idea of Developing a DP Algorithm

Step1: Structure: Characterize the structure of an optimal solution.

- Decompose the problem into smaller problems, and find a relation between the structure of the optimal solution of the original problem and the solutions of the smaller problems.

Step2: Principle of Optimality: Recursively define the value of an optimal solution.

- Express the solution of the original problem in terms of optimal solutions for smaller problems.

The Idea of Developing a DP Algorithm

Step 3: Bottom-up computation: Compute the value of an optimal solution in a bottom-up fashion by using a table structure.

Step 4: Construction of optimal solution: Construct an optimal solution from computed information.

Steps 3 and 4 may often be combined.

Remarks on the Dynamic Programming Approach

- Steps 1-3 form the basis of a dynamic-programming solution to a problem.
- Step 4 can be omitted if only the value of an optimal solution is required.

Developing a DP Algorithm for Knapsack

Step 1: Decompose the problem into smaller problems.

We construct an array $V[0..n, 0..W]$.

For $1 \leq i \leq n$, and $0 \leq w \leq W$, the entry $V[i, w]$ will store the maximum (combined) computing time of any subset of files $\{1, 2, \dots, i\}$ of (combined) size at most w .

If we can compute all the entries of this array, then the array entry $V[n, W]$ will contain the maximum computing time of files that can fit into the storage, that is, the solution to our problem.

Developing a DP Algorithm for Knapsack

Step 2: Recursively define the value of an optimal solution in terms of solutions to smaller problems.

Initial Settings: Set

$$\begin{aligned} V[0, w] &= 0 && \text{for } 0 \leq w \leq W, && \text{no item} \\ V[i, w] &= -\infty && \text{for } w < 0, && \text{illegal} \end{aligned}$$

Recursive Step: Use

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

for $1 \leq i \leq n, 0 \leq w \leq W$.

Correctness of the Method for Computing $V[i, w]$

Lemma: For $1 \leq i \leq n$, $0 \leq w \leq W$,

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

Proof: To compute $V[i, w]$ we note that we have only two choices for file i :

Leave file i : The best we can do with files $\{1, 2, \dots, i - 1\}$ and storage limit w is $V[i - 1, w]$.

Take file i (only possible if $w_i \leq w$): Then we gain v_i of computing time, but have spent w_i bytes of our storage. The best we can do with remaining files $\{1, 2, \dots, i - 1\}$ and storage $(w - w_i)$ is $V[i - 1, w - w_i]$.

Totally, we get $v_i + V[i - 1, w - w_i]$.

Note that if $w_i > w$, then $v_i + V[i - 1, w - w_i] = -\infty$ so the lemma is correct in any case.

Developing a DP Algorithm for Knapsack

Step 3: Bottom-up computing $V[i, w]$ (using iteration, not recursion).


Bottom: $V[0, w] = 0$ for all $0 \leq w \leq W$.

Bottom-up computation: Computing the table using

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

row by row.

| $V[i, w]$ | $w=0$ | 1 | 2 | 3 | ... | ... | W |
|-----------|-------|---|---|---|-----|-----|-----|
| $i=0$ | 0 | 0 | 0 | 0 | ... | ... | 0 |
| 1 | → | | | | | | |
| 2 | → | | | | | | |
| ⋮ | → | | | | | | |
| n | → | | | | | | |



bottom

up

Example of the Bottom-up computation

Let $W = 10$ and

| | | | | |
|-------|----|----|----|----|
| i | 1 | 2 | 3 | 4 |
| v_i | 10 | 40 | 30 | 50 |
| w_i | 5 | 4 | 6 | 3 |

| $V[i, w]$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|-----------|---|---|---|----|----|----|----|----|----|----|----|
| $i = 0$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 10 | 10 | 10 | 10 | 10 | 10 |
| 2 | 0 | 0 | 0 | 0 | 40 | 40 | 40 | 40 | 40 | 50 | 50 |
| 3 | 0 | 0 | 0 | 0 | 40 | 40 | 40 | 40 | 40 | 50 | 70 |
| 4 | 0 | 0 | 0 | 50 | 50 | 50 | 50 | 90 | 90 | 90 | 90 |

Remarks:

- The final output is $V[4, 10] = 90$.
- The method described does not tell which subset gives the optimal solution. (It is $\{2, 4\}$ in this example).

The Dynamic Programming Algorithm

```
KnapSack( $v, w, n, W$ )
{
  for ( $w = 0$  to  $W$ )  $V[0, w] = 0$ ;
  for ( $i = 1$  to  $n$ )
    for ( $w = 0$  to  $W$ )
      if ( $w[i] \leq w$ )
         $V[i, w] = \max\{V[i - 1, w], v[i] + V[i - 1, w - w[i]]\}$ ;
      else
         $V[i, w] = V[i - 1, w]$ ;
  return  $V[n, W]$ ;
}
```

Time complexity: Clearly, $O(nW)$.

Constructing the Optimal Solution

- The algorithm for computing $V[i, w]$ described in the previous slide does not keep record of which subset of items gives the optimal solution.
- To compute the actual subset, we can add an auxiliary boolean array $keep[i, w]$ which is 1 if we decide to take the i -th file in $V[i, w]$ and 0 otherwise.

Question: How do we use all the values $keep[i, w]$ to determine the subset T of files having the maximum computing time?

Constructing the Optimal Solution

Question: How do we use the values $keep[i, w]$ to determine the subset T of items having the maximum computing time?

If $keep[n, W]$ is 1, then $n \in T$. We can now repeat this argument for $keep[n - 1, W - w_n]$.

If $keep[n, W]$ is 0, the $n \notin T$ and we repeat the argument for $keep[n - 1, W]$.

Therefore, the following partial program will output the elements of T :

```
 $K = W;$ 
for ( $i = n$  downto 1)
  if ( $keep[i, K] == 1$ )
  {
    output  $i$ ;
     $K = K - w[i]$ ;
  }
```


The Complete Algorithm for the Knapsack Problem

```
KnapSack( $v, w, n, W$ )
{
  for ( $w = 0$  to  $W$ )  $V[0, w] = 0$ ;
  for ( $i = 1$  to  $n$ )
    for ( $w = 0$  to  $W$ )
      if ( $(w[i] \leq w)$  and  $(v[i] + V[i - 1, w - w[i]] > V[i - 1, w])$ )
        {
           $V[i, w] = v[i] + V[i - 1, w - w[i]]$ ;
           $keep[i, w] = 1$ ;
        }
      else
        {
           $V[i, w] = V[i - 1, w]$ ;
           $keep[i, w] = 0$ ;
        }
    }
   $K = W$ ;
  for ( $i = n$  downto 1)
    if ( $keep[i, K] == 1$ )
      {
        output  $i$ ;
         $K = K - w[i]$ ;
      }
  return  $V[n, W]$ ;
}
```