COMP20007 Design of Algorithms

Dynamic Programming Part 2: Knapsack Problem

Daniel Beck

Lecture 20

Semester 1, 2020

Knapsack

item	weight	value
1	2	\$12
2	1	\$10
3	3	\$20
4	2	\$15

capacity = 5

2^n

#	weight	value
(1)	2	12
(2)	1	10
(3)	3	20
(4)	2	15
(1,2)	3	22
(1,3)	5	32
(1,4)	4	27
(2,3)	4	30
(2,4)	3	25
(3,4)	5	35
(1,2,3)	6	42
(1,2,4)	5	37
(1,3,4)	7	47
(2,3,4)	6	45
(1,2,3,4)	8	57

KNAPSACK = F

F((1,2), 5) = (1,2)

F((1,2,3), 5) = (1,3)

$\ne$

n item	weight	value
1	2	\$12
2	1	\$10
3	3	\$20
4	2	\$15

capacity = 5 = W

iterate over items $i = 0 \cdots n$

iterate over capacity $j = 0 \cdots W$

now KNAPSACK $K = F(i, j)$

max value $F(n, w)$

base case: $F(0,1) = 0$, $F(1,0) = 0$

0 item. 1 capacity   1 item, 0 capacity

find $F(i,j) = ?\ F(i-1, j)$

if $w_i > j$: $F(i,j) = F(i-1, j)$

else $w_i \le j$:

if $i \notin$ solution: $F(i,j) = F(i-1, j)$

else $i \in$ solution $F(i,j) = \underbrace{V_i}_{\text{value of } i} + F(i-1, \underbrace{j-W_i}_{\text{capacity} - w_i})$

The Knapsack Problem

Given n items with

- weights: $w_1, w_2, \ldots, w_n$
- values: $v_1, v_2, \ldots, v_n$
- knapsack of capacity W

find the most valuable selection of items that will fit in the knapsack.

The Knapsack Problem

Given n items with

- weights: $w_1, w_2, \ldots, w_n$
- values: $v_1, v_2, \ldots, v_n$
- knapsack of capacity W

find the most valuable selection of items that will fit in the knapsack.

We assume that all entities involved are positive integers.

item	weight	value
1	2	$12
2	1	$10
3	3	$20
4	2	$15

CAPACITY = 5

capacity →

item i ↓ \ j	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15	25	30	37

fail for capacity 1 & 2 since $w_3 = 3$ (can't fit)

(10+15) vs. 22.

(32 vs 22+15)

F(4,5)
end of algorithm
F(n, m).

to look at what item is included.
(subset of item)

backtrack

① F(4,5) > F(3,5) (item 4 is include)

{4,

② F(4−1, 5−w_4) → F(3,3).
(2.)

compare F(3,3) & F(2,3)

F(2,3) = F(3,3)

(item 3 is not included)

③ look at item 2

F(2,3) ~~vs~~ F(1,3)

item 2 is included

{4, 2

④ look at item 1

F(2−1, 3−w_2) → F(1,2)

F(1,2) > F(0,2) item 1 is include

{4, 2, 1}

Example 2: The Knapsack Problem

Express the solution recursively:

$$K(i, j) = 0 \text{ if } i = 0 \text{ or } j = 0$$

Otherwise:

$$K(i, j) = \begin{cases} max(K(i-1, j), K(i-1, j-w_i) + v_i) & \text{if } j \geq w_i \\ K(i-1, j) & \text{if } j < w_i \end{cases}$$

Example 2: The Knapsack Problem

Express the solution recursively:

$$K(i,j) = 0 \text{ if } i = 0 \text{ or } j = 0$$

Otherwise:

$$K(i,j) = \begin{cases} max(K(i-1,j), K(i-1,j-w_i)+v_i) & \text{if } j \geq w_i \\ K(i-1,j) & \text{if } j < w_i \end{cases}$$

For a bottom-up solution we need to write the code that systematically fills a two-dimensional table.

The table will have $n+1$ rows and $W+1$ columns.

Example 2: The Knapsack Problem

function $\textsc{Knapsack}(v[1..n], w[1..n], W)$

Initalize the first row and col to be 0

for $i \leftarrow 0$ to n **do** $K[i, 0] \leftarrow 0$

for $j \leftarrow 1$ to W **do** $K[0, j] \leftarrow 0$

for $i \leftarrow 1$ to n **do**

  for $j \leftarrow 1$ to W **do**

    if $j < w_i$ **then**

      $K[i, j] \leftarrow K[i-1, j]$

    else

      $K[i, j] \leftarrow max(K[i-1, j], K[i-1, j-w_i] + v_i)$

return $K[n, W]$

return the last element

Solving the Knapsack Problem with Memoing

- To some extent the bottom-up (table-filling) solution is overkill: It finds the solution to every conceivable sub-instance.

Solving the Knapsack Problem with Memoing

- To some extent the bottom-up (table-filling) solution is overkill: It finds the solution to every conceivable sub-instance.
- Most entries cannot actually contribute to a solution.

Solving the Knapsack Problem with Memoing

- To some extent the bottom-up (table-filling) solution is overkill: It finds the solution to every conceivable sub-instance.
- Most entries cannot actually contribute to a solution.
- In this situation, a top-down approach, with $\boxed{\text{memoing}}$, is preferable.

?

Solving the Knapsack Problem with Memoing

- To some extent the bottom-up (table-filling) solution is overkill: It finds the solution to every conceivable sub-instance.
- Most entries cannot actually contribute to a solution.
- In this situation, a top-down approach, with memoing, is preferable.
- To keep the memo table small, make it a hash table.

Solving the Knapsack Problem with Memoing

function $\text{KNAP}(i,j)$ ▷ Uses a global hashtable

  if $i = 0$ or $j = 0$ **then**

    return 0

  if key (i, j) is in hashtable **then**

    return the corresponding value (that is, $K(i, j)$)

call value directly instead of counting again

  if $j < w_i$ **then**

    $k \leftarrow \text{KNAP}(i - 1, j)$

  else

    $k \leftarrow max(\text{KNAP}(i - 1, j), \text{KNAP}(i - 1, j - w_i) + v_i)$

  insert k into hashtable, with key (i, j)

  return k

Knapsack - Complexity

- Time (and space) complexity of Knapsack is $\Theta(nW)$

Item capacity

Knapsack - Complexity

- Time (and space) complexity of Knapsack is $\Theta(nW)$
- This is called pseudopolynomial time: the algorithm is polynomial in the value of the input, not its length.
  - Counting Sort is another example. $\Theta(n + \underbrace{W}_{\text{largest element in array}})$

Knapsack - Complexity

insertion sort $\Theta(n^2)$

32 bit integer $\Theta((32b)^2) = \Theta(1024b^2) = \Theta(b^2)$

but all int has constant bit length

- Time (and space) complexity of Knapsack is $\Theta(nW)$
- This is called pseudopolynomial time: the algorithm is polynomial in the value of the input, not its length.
  - Counting Sort is another example.
- Pseudopolynomial is not in general polynomial because it is exponential in the number of bits.

since it is related to the value of the input, so the representation matters

$W = 11110$

length 5

value 30

if 111110

length 6

value 62

value $O(2^l)$

exponential algorithm

Summary

- Dynamic Programming recipe:

Summary

- Dynamic Programming recipe:
  - Split into subproblems.
  - Solutions overlap. → store solution and reuse.

Summary

- Dynamic Programming recipe:
  - Split into subproblems.
  - Solutions overlap.
- A DP solution for Knapsack results in a pseudopolynomial time algorithm.

Summary

- Dynamic Programming recipe:
  - Split into subproblems.
  - Solutions overlap.
- A DP solution for Knapsack results in a pseudopolynomial time algorithm.
  - Uses two “variables” to split the problem.
  - Can use memoing to speed it up.

?

Summary

- Dynamic Programming recipe:
  - Split into subproblems.
  - Solutions overlap.
- A DP solution for Knapsack results in a pseudopolynomial time algorithm.
  - Uses two "variables" to split the problem.
  - Can use memoing to speed it up.

global data structure that can be use for each recursive function

Next lecture: the last one! (before the revision)

Summary

- Dynamic Programming recipe:
  - Split into subproblems.
  - Solutions overlap.
- A DP solution for Knapsack results in a pseudopolynomial time algorithm.
  - Uses two "variables" to split the problem.
  - Can use memoing to speed it up.

Next lecture: the last one! (before the revision)

$$\mathrm{P} = \mathrm{NP}?$$