بسم اللّه الرّحمن الرّحیم

دانشگاه صنعتی اصفهان ـ دانشکدهٔ مهندسی برق و کامپیوتر

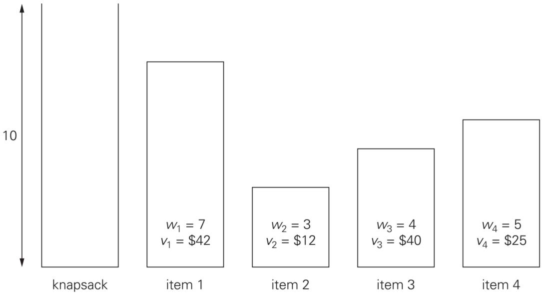
(نیم‌سال تحصیلی ۴۰۰۱)

طراحی الگوریتم‌ها

حسین فلسفین

0-1 Knapsack Problem

Given n items of known weights $w_1, w_2, \ldots, w_n$ and values $v_1, v_2, \ldots, v_n$ and a knapsack of capacity W, find the most valuable subset of the items that fit into the knapsack.



(a)

Subset	Total weight	Total value
∅	0	$ 0
{1}	7	$42
{2}	3	$12
{3}	4	$40
{4}	5	$25
{1, 2}	10	$54
{1, 3}	11	not feasible
{1, 4}	12	not feasible
{2, 3}	7	$52
{2, 4}	8	$37
{3, 4}	**9**	**$65**
{1, 2, 3}	14	not feasible
{1, 2, 4}	15	not feasible
{1, 3, 4}	16	not feasible
{2, 3, 4}	12	not feasible
{1, 2, 3, 4}	19	not feasible

(b)

NP-hardness :(

The exhaustive-search approach to this problem leads to generating all the subsets of the set of n items given, computing the total weight of each subset in order to identify feasible subsets (i.e., the ones with the total weight not exceeding the knapsack capacity), and finding a subset of the largest value among them. Since the number of subsets of an n-element set is 2^n, the exhaustive search leads to a $\Omega(2^n)$ algorithm, no matter how efficiently individual subsets are generated.

This problems is one of the best-known examples of socalled NP-hard problems. No polynomial-time algorithm is known for any NP-hard problem. Moreover, most computer scientists believe that such algorithms do not exist, although this very important conjecture has never been proven.

A Dynamic Programming Approach to the 0-1 Knapsack Problem

We assume here that all the weights and the knapsack capacity are positive integers; the item values do not have to be integers.

To design a dynamic programming algorithm, we need to derive a recurrence relation that expresses a solution to an instance of the knapsack problem in terms of solutions to its smaller subinstances. Let us consider an instance defined by the first i items, $1 \leq i \leq n$, with weights $w_1, w_2, \ldots, w_i$, values $v_1, v_2, \ldots, v_i$, and knapsack capacity j, $1 \leq j \leq W$. Let $F(i, j)$ be the value of an optimal solution to this instance, i.e., the value of the most valuable subset of the first i items that fit into the knapsack of capacity j. We can divide all the subsets of the first i items that fit the knapsack of capacity j into two categories: those that do not include the ith item and those that do.

1. *Among the subsets that do not include the ith item, the value of an optimal subset is, by definition, $F(i-1, j)$.*

2. *Among the subsets that do include the ith item (hence, $j \geq w_i$), an optimal subset is made up of this item and an optimal subset of the first $i-1$ items that fits into the knapsack of capacity$j - w_i$. The value of such an optimal subset is $v_i + F(i-1, j-w_i)$.*

Thus, the value of an optimal solution among all feasible subsets of the first i items is the maximum of these two values. Of course, if the ith item does not fit into the knapsack, the value of an optimal subset selected from the first i items is the same as the value of an optimal subset selected from the first $i-1$ items.

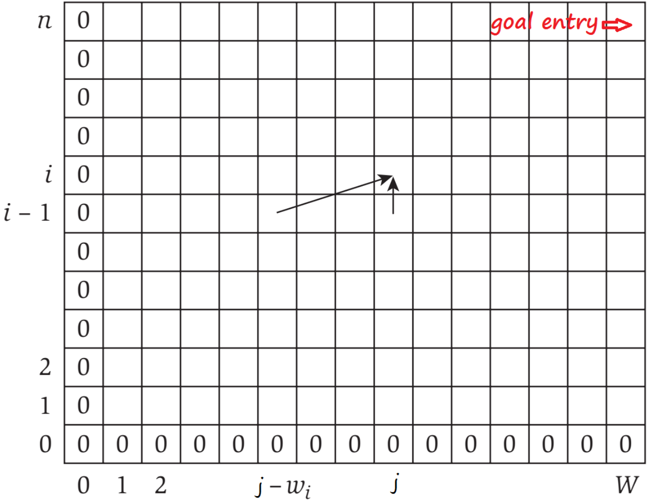
$$F(i,j) = \begin{cases} \max\{F(i-1,j), v_i + F(i-1, j-w_i)\}, & \textbf{if } j \geq w_i, \\ F(i-1,j), & \textbf{if } j < w_i. \end{cases}$$

It is convenient to define the initial conditions as follows:

$$F(0,j) = 0 \textbf{ for } j \geq 0 \textbf{ and } F(i,0) = 0 \textbf{ for } i \geq 0.$$

Our goal is to find $F(n, W)$, the maximal value of a subset of the n given items that fit into the knapsack of capacity W, and an optimal subset itself. For $i, j > 0$, to compute the entry in the ith row and the jth column, $F(i, j)$, we compute the maximum of the entry in the previous row and the same column and the sum of v_i and the entry in the previous row and w_i columns to the left. The table can be filled either row by row or column by column.

		0	$j-w_i$	j	W
	0	0	0	0	0
	$i-1$	0	$F(i-1, j-w_i)$	$F(i-1, j)$	
w_i, v_i	i	0		$F(i, j)$	
	n	0			goal


goal entry
n
i
i − 1
2
1
0
0 1 2
j − w_i
j
W
0

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

capacity $W = 5$.

capacity j

	i	0	1	2	3	4	5
	0	0	0	0	0	0	0
$w_1 = 2,\ v_1 = 12$	1	0	0	12	12	12	12
$w_2 = 1,\ v_2 = 10$	2	0	10	12	22	22	22
$w_3 = 3,\ v_3 = 20$	3	0	10	12	22	30	32
$w_4 = 2,\ v_4 = 15$	4	0	10	15	25	30	**37**

We can find the composition of an optimal subset by backtracing the computations of this entry in the table. Since $F(4,5) > F(3,5)$, item 4 has to be included in an optimal solution along with an optimal subset for filling $5-2=3$ remaining units of the knapsack capacity. The value of the latter is $F(3,3)$. Since $F(3,3) = F(2,3)$, item 3 need not be in an optimal subset. Since $F(2,3) > F(1,3)$, item 2 is a part of an optimal selection, which leaves element $F(1, 3-1)$ to specify its remaining composition. Similarly, since $F(1,2) > F(0,2)$, item 1 is the final part of the optimal solution {item 1, item 2, item 4}.

The time efficiency and space efficiency of this algorithm are both in $\Theta(nW)$. The time needed to find the composition of an optimal solution is in $O(n)$.

NP-hardness!? $\Theta(nW)$!?

The fact that the previous expression for the number of array entries computed is linear in n can mislead one into thinking that the algorithm is efficient for all instances containing n items. This is not the case. The other term in that expression is W, and there is no relationship between n and W. Therefore, for a given n, we can create instances with arbitrarily large running times by taking arbitrarily large values of W. For example, the number of entries computed is in $\Theta(n \times n!)$ if W equals $n!$. If $n = 20$ and $W = 20!$, the algorithm will take thousands of years to run on a modern-day computer. When W is extremely large in comparison with n, this algorithm is worse than the brute-force algorithm that simply considers all subsets.

الگوریتم فوق برای مسئلهٔ کوله‌پشتی ۰ ــ ۱ را یک الگوریتم *pseudo-polynomial time* گویند. مسائل *NP-hard*ی که برای آنها یک الگوریتم *pseudo-polynomial time* وجود دارد را معمولاً *weakly NP-hard* می‌نامند. به گزارهٔ مهم زیر دقت کنید:

A pseudo-polynomial time algorithm will display ‘exponential behavior’ only when confronted with instances containing ‘exponentially large’ numbers, which might be rare for the application we are interested in. If so, this type of algorithm might serve our purposes almost as well as a polynomial time algorithm.

Memory Functions

Dynamic programming deals with problems whose solutions satisfy a recurrence relation with overlapping subproblems. The direct top-down approach to finding a solution to such a recurrence leads to an algorithm that solves common subproblems more than once and hence is very inefficient (typically, exponential or worse). The classic dynamic programming approach, on the other hand, works bottom up: it fills a table with solutions to all smaller subproblems, but each of them is solved only once. An unsatisfying aspect of this approach is that solutions to some of these smaller subproblems are often not necessary for getting a solution to the problem given. Since this drawback is not present in the top-down approach, it is natural to try to combine the strengths of the top-down and bottom-up approaches.

The goal is to get a method that solves only subproblems that are necessary and does so only once. Such a method exists; it is based on using memory functions. This method solves a given problem in the top-down manner but, in addition, maintains a table of the kind that would have been used by a bottom-up dynamic programming algorithm. Initially, all the table's entries are initialized with a special "null" symbol to indicate that they have not yet been calculated. Thereafter, whenever a new value needs to be calculated, the method checks the corresponding entry in the table first: if this entry is not "null," it is simply retrieved from the table; otherwise, it is computed by the recursive call whose result is then recorded in the table.

ALGORITHM *MFKnapsack*(i, j)

```
//Implements the memory function method for the knapsack problem
//Input: A nonnegative integer i indicating the number of the first
//        items being considered and a nonnegative integer j indicating
//        the knapsack capacity
//Output: The value of an optimal feasible subset of the first i items
//Note: Uses as global variables input arrays Weights[1..n], Values[1..n],
//and table F[0..n, 0..W] whose entries are initialized with −1's except for
//row 0 and column 0 initialized with 0's
if F[i, j] < 0
    if j < Weights[i]
        value ← MFKnapsack(i − 1, j)
    else
        value ← max(MFKnapsack(i − 1, j),
                    Values[i] + MFKnapsack(i − 1, j − Weights[i]))
    F[i, j] ← value
return F[i, j]
```

	i	**capacity** j: 0	1	2	3	4	5
	0	0	0	0	0	0	0
$w_1 = 2,\ v_1 = 12$	1	0	0	12	12	12	12
$w_2 = 1,\ v_2 = 10$	2	0	—	12	22	—	22
$w_3 = 3,\ v_3 = 20$	3	0	—	—	22	—	32
$w_4 = 2,\ v_4 = 15$	4	0	—	—	—	—	**37**

After initializing the table, the recursive function needs to be called with $i = n$ (the number of items) and $j = W$ (the knapsack capacity). Only 11 out of 20 nontrivial values (i.e., not those in row 0 or in column 0) have been computed.

Chained Matrix Multiplication

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 7 & 8 & 9 & 1 \\ 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 \end{bmatrix} = \begin{bmatrix} 29 & 35 & 41 & 38 \\ 74 & 89 & 104 & 83 \end{bmatrix}$$

Because there are $2 \times 4 = 8$ ***entries in the product, the total number of elementary multiplication is*** $2 \times 4 \times 3$***.***

In general, to multiply an $i \times j$ ***matrix times a*** $j \times k$ ***matrix using the standard method, it is necessary to do*** $i \times j \times k$ ***elementary multiplications.***

$$\begin{array}{cccccccc} A & \times & B & \times & C & \times & D \\ 20\times 2 & & 2\times 30 & & 30\times 12 & & 12\times 8 \end{array}$$

$$\begin{array}{ll} A\,(B\,(CD)) & 30\times 12\times 8+\ \ 2\times 30\times\ \ 8+20\times\ \ 2\times 8=\ \ 3,680 \\ (AB)\,(CD) & 20\times 2\times 30+30\times 12\times\ \ 8+20\times 30\times 8=\ \ 8,880 \\ A\,((BC)\,D) & 2\times 30\times 12+\ \ 2\times 12\times\ \ 8+20\times\ \ 2\times 8=\ \ 1,232 \\ ((AB)\,C)\,D & 20\times 2\times 30+20\times 30\times 12+20\times 12\times 8=10,320 \\ (A\,(BC))\,D & 2\times 30\times 12+20\times\ \ 2\times 12+20\times 12\times 8=\ \ 3,120 \end{array}$$

The third order is the optimal order for multiplying the four matrices.

Our goal is to develop an algorithm that determines the optimal order for multiplying n matrices. The optimal order depends only on the dimensions of the matrices. Therefore, besides n, these dimensions would be the only input to the algorithm.

The brute-force algorithm

> *The brute-force algorithm is to consider all possible orders and take the minimum. We will show that this algorithm is at least exponential-time.*

Let t_n be the number of different orders in which we can multiply n matrices: $A_1, A_2, \ldots, A_n$. A subset of all the orders is the set of orders for which A_1 is the last matrix multiplied. The number of different orders in this subset is t_{n-1}, because it is the number of different orders with which we can multiply A_2 through A_n:

$$A_1(\underbrace{A_2A_3\cdots A_n}_{\substack{t_{n-1}\text{ different}\\ \text{orders}}})$$

A second subset of all the orders is the set of orders for which A_n is the last matrix multiplied. Clearly, the number of different orders in this subset is also t_{n-1}.

Therefore,

$$t_n \geq t_{n-1} + t_{n-1} = 2t_{n-1}.$$

Because there is only one way to multiply two matrices, $t_2 = 1$*.*
It can readily be shown that $t_n \geq 2^{n-2}$*.*

Catalan numbers

$$\begin{array}{ccc}
(A_1)(A_2A_3\cdots A_n) & \rightarrow & t_1\times t_{n-1}\\
(A_1A_2)(A_3A_4\cdots A_n) & \rightarrow & t_2\times t_{n-2}\\
(A_1A_2A_3)(A_4A_5\cdots A_n) & \rightarrow & t_3\times t_{n-3}\\
\vdots & \vdots & \vdots\\
(A_1A_2\cdots A_{n-1})(A_n) & \rightarrow & t_{n-1}\times t_1
\end{array}$$

$$t_n=\sum_{k=1}^{n-1}t_kt_{n-k}$$

It can be proved by induction that: $t_n=\frac{1}{n}\binom{2(n-1)}{n-1}$

The n***th Catalan number:*** $C_n=\frac{1}{n+1}\binom{2n}{n}$

We have: $t_n=C_{n-1}$

It can be shown that: $C_n\approx\frac{4^n}{\sqrt{\pi n^3}}$ ☞ رشدش بد است!