Deriving Dynamic Programming via Thinning and Incrementalization

Akimasa Morihata (Univ. Tokyo)

Masato Koishi (Toshiba Solutions Corp.)

Atsushi Ohori (RIEC, Tohoku Univ.)

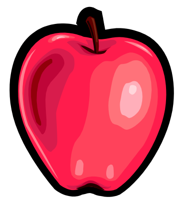
FLOPS 2014, Kanazawa, Japan

0-1 Knapsack Problem

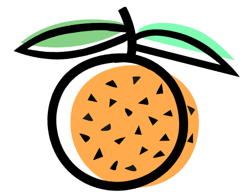
Given: budget $\boldsymbol{B}$ and items $\boldsymbol{i}_{1,\ldots,}\boldsymbol{i}_n$

Objective: most valuable itemset that can buy

$\boldsymbol{B}$ **= ¥300**

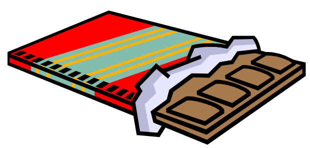


0.5MJ, ¥120



0.2MJ, ¥60

1.6MJ, ¥150



1.3MJ, ¥90

1.9MJ, ¥140

Textbook Solution: Dynamic Programming

1. *Recurrence equation* (*principle of optimality*):

$$\boldsymbol{opt(S \uplus \{i\}, c)}$$
$$\boldsymbol{= \max \{opt(S, c), value(i) + opt(S, c - cost(i))\}}$$

2. *Table-filling implementation*:

	$c = 0$	$c = 1$	…	$c = cost(i_1)+1$	…	$c = B$
$\{i_0\}$						
$\{i_0, i_1\}$		$+value(i_1)$				
…						
S						

Textbook Approach to DP

Problem Description

↓ Thinning

Recurrence Equation

↓ Incrementalization

Table-filling Implementation

Our Contribution

New method of systematically developing DP by combining known methods:

- Thinning (Bird & de Moor ’96; Morihata ’11) to expose a recurrence equation
- Incrementalization (Liu+ ’03; ’05) to develop efficient table-filling computation

This is (to our knowledge) a first study on their cooperation

Problem Description

$$knapsack = max_{\leq value} \circ feasible \circ subsets$$

maximizer filter enumerator

$$subsets\ [\,] = \{[\,]\}$$

$$subsets\ (i : is) = \textbf{let}\ r = subsets\ is\ \textbf{in}\ r \cup map\ (i:)\ r$$

$$feasible = filter\ (\lambda is.\ cost\ is \leq B)$$

$$value = sum \circ map\ valueOfItem$$

$$cost = sum \circ map\ costOfItem$$

Thinning by Shortcut Fusion

Thinning (Bird & de Moor '97): avoid exhaustive enumeration by fusing maximizer/filter/enumerator

Theorem. (refinement of Morihata'11)

$$\begin{aligned}
&\boldsymbol{max}_{\preceq}\ (\boldsymbol{filter}\ \boldsymbol{p}\ (\boldsymbol{gen}\ (\cup)\ (\lambda \boldsymbol{a}.\ \boldsymbol{map}\ (\boldsymbol{a}{:}))\ (\{[\,]\})))\\
&\quad = \boldsymbol{max}_{\preceq}\ (\boldsymbol{gen}\ (\lambda x\ y.\ \ \boldsymbol{max}_{\preceq \cap \gg}\ (x \cup y))\\
&\qquad\qquad\qquad (\lambda \boldsymbol{a}.\ \boldsymbol{max}_{\preceq \cap \gg} \circ \boldsymbol{filter}\ \boldsymbol{p} \circ \boldsymbol{map}\ (\boldsymbol{a}{:}))\\
&\qquad\qquad\qquad (\boldsymbol{filter}\ \boldsymbol{p}\ \{[\,]\}))
\end{aligned}$$

discard useless ones

if $\boldsymbol{p}\ \boldsymbol{x} = \boldsymbol{h}\ \boldsymbol{x} \ll \mathbf{c}$,

$\preceq$ and $\gg$ are monotone on $(:)$, $\gg$ is increasing on $(:)$,

and $\boldsymbol{gen}$'s type is $\forall\beta.\ (\beta\to\beta\to\beta)\to(A\to\beta\to\beta)\to\beta\to\beta$.

Applying Thinning to ***knapsack***

$$knapsack = max_{\leq value} \circ feasible \circ subsets$$

$$subsets\ [\,] = \{[\,]\}$$

$$subsets\ (i : is) = \text{let } r = subsets\ is \text{ in } r \cup map\ (i{:})\ r$$

Thinning

$$knapsack = max_{\leq value} \circ subsets'$$

$$subsets'\ [\,] = \{[\,]\}$$

$$subsets'\ (i : is)$$
$$= \text{let } r = subsets'\ is$$
$$\text{in } max_{\leq value \cap \geq cost}\ (r \cup feasible\ (map\ (i{:})\ r))$$

Note: Recurrence Equation Exposed

$\boldsymbol{max}_{\leq value \cap \geq cost}$ removes an element if a more valuable and cheaper one exists, namely,†

$$\boldsymbol{max}_{\leq value \cap \geq cost}\, S = \{\boldsymbol{max}_{\leq value}\, \{is' \mid is' \in S,\ is' \leq_{cost} is\} \mid is \in S\}$$

This essentially leads to the recurrence equation:

$$opt(S \uplus \{i\}) = \{\boldsymbol{max}_{\leq value}\, \{is \mid is' \in opt(S),\ is \in \{is', i : is'\},\ cost(is) \leq c\} \mid 0 \leq c \leq B\}$$

†assume no two candidates have the same value

Drawback of Thinning

Obtained program is not very efficient

$$knapsack = max_{\leq value} \circ subsets'$$

$$subsets'\ [] = \{[]\}$$

$$\begin{aligned} &subsets'\ (i : is) \\ &\quad = \textbf{let}\ r = subsets'\ is \\ &\qquad \textbf{in}\ max_{\leq value \cap \geq cost}\ (r \cup feasible\ (map\ (i{:})\ r)) \end{aligned}$$

$O(B)$ candidates

$O(B^2)$ time?

Incrementalization

Incremenalization (Liu+ '03; '05):
improving efficiency by reusing previous results

Theorem. (well-known)
If $s_1 \subseteq \cdots \subseteq s_m$, for associative and commutative $\oplus$,

$$[\bigoplus s_m, \ldots, \bigoplus s_1]$$
$$= \mathit{foldr}\ f\ [\bigoplus s_1]\ [s_m \setminus s_{m-1}, \ldots, s_2 \setminus s_1]$$
$$\text{where } f\ s\ (r{:}rs) = (\bigoplus s) \oplus r : r : rs$$

Incrementalize *max*

$$max_{\leq value \cap \geq cost}\ S$$
$$= \{max_{\leq value}\ \{is' \mid is' \in S,\ is' \leq_{cost} is\} \mid is \in S\}$$

let $S = \{is_1, \ldots, is_m\}$ s.t. $is_1 \leq_{cost} \cdots \leq_{cost} is_m$

$$max_{\leq value \cap \geq cost}\ S$$
$$= \{max_{\leq value}\ \{is_1, \ldots, is_k\} \mid 1 \leq k \leq m\}$$

Incrementalization

$O(m)$ time

$$max_{\leq value \cap \geq cost}\ S\ = foldr\ f\ [is_1]\ [is_m\ , \ldots, is_2]$$
$$\textbf{where}\ f\ is\ (r{:}rs) = max_{\leq value}\ \{is, r\} : r : rs$$

Obtained Program

$$knapsack = max_{\leq value} \circ subsets'$$

$$subsets'\ [\,] = \{[\,]\}$$

$$\begin{aligned} subsets'\ (i : is) &\\ = \text{let } r = subsets'\ is &\\ \text{in } max_{\leq value \cap \geq cost}\ (r \cup feasible\ (map\ (i{:})\ r)) & \end{aligned}$$

O(*B*) time

O(*B*) candidates

- Time complexity: $\mathbf{O}(nB)$
- It is essentially the one presented by de Moor (1995)

What Written in the Paper

- Our approach scales to other examples (derivations are more complicated, though)
  - longest common sequence problem
  - association-rule mining on two-dimensional numeric data (Fukuda+'99; 01)
- Discussion on possibility of automation
  - Difficulty:
    bridging the gap between two methods

Related Work (1)

- Thinning (Bird & de Moor’ 97)
  - useful to deriving DP from naive specification
  - drawback: results may be too abstract, far from efficient implementation
- Incrementalization (Liu+ ’03; ’05)
  - useful for deriving efficient table-filling implementation from recurrence equations

Our observation: they are *orthogonal*

Related Work (2)

- de Moor (1995):
  - proposed a law to derive DP for a class of optimization problems
  - Result for the knapsack problem is almost the same
  - We modularized his law into two components: thinning and incrementalization
    ➔ more problems can be solved

Conclusion & Future Work

Combination of thinning and incrementalization is useful for developing DP!

Future work:

- Automation
  - Domain-specific? Solver-based?
- Combination of other techniques