Dynamic Programming

via

Static Incrementalization

Y. Annie Liu

Computer Science Department
Indiana University

Dynamic Programming

- advanced algorithm design technique
  often used for optimization problems.

- in general,
  for problems that require recursively solving subproblems.

  subproblems overlap $\longrightarrow$
  straightforward recursion solves them repeatedly,
  often takes exponential time.

- solution: compute each subproblem once and save its result.
  classical techniques: memoization or tabulation

Example: longest common subsequence

Longest len. of common subsequences of $x_1, ..., x_m$ and $y_1, ..., y_n$.

$$\underset{i,j \geq 0}{c(i,j)} = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c(i-1, j-1) + 1 & \text{if } i,j \neq 0 \text{ and } x_i = y_j \\ \max(c(i, j-1),\ c(i-1, j)) & \text{otherwise} \end{cases}$$

Straightforward recursive program:

$$\boxed{\begin{array}{l} c(i,j) \quad \text{where } i \geq 0 \text{ and } j \geq 0 \\ \triangleq \quad \textbf{if } i = 0 \lor j = 0 \textbf{ then } 0 \\ \quad\;\; \textbf{else if } x[i] = y[j] \textbf{ then } c(i-1, j-1) + 1 \\ \quad\;\; \textbf{else } \max(c(i, j-1),\ c(i-1, j)) \end{array}}$$

Memoization and Tabulation

<u>Memoization</u>: use a mechanism separate from the original prog.

- advantage:
  original recursive program needs no change.
  only values needed for the solution are computed.

- disadvantage:
  separate table management has interpretive overhead.
  a general strategy can not be efficient for all problems.

<u>Tabulation</u>: build table management into an optimized prog.

- advantage:
  compiled table management, no interpretive overhead
  specialized computation strategy, also optimized space usage.

- disadvantage:
  manually rewrite original program.
  may compute values not needed for the solution.

Incremental Programs

f' is an <u>incremental version</u> of f under $\oplus$

if f' computes $f(x \oplus y)$ efficiently by making use of f(x).

(x', r) x' r

Examples:

- f is $sort$, x is a list of numbers, $x \oplus i$ is $cons(i, x)$.

  if $sort(x) = r$ and $cons(i, x) = x'$, then $sort'(x', r) = sort(x')$.

- f is C compiler, x is C program, $\oplus$ is change to program.

- f is loop body, x is ind. variable, $\oplus$ is increment to ind. var.

$x = prev(x')$.

Static Incrementalization

- efficient computation proceeds in incremental fashion:
  compute one iteration using results of previous iteration.

3 tasks:

I. identify expensive f, $\oplus$;
II. obtain f';
III. form optimized f.

3 steps:

II { 1. cache; 2. incrementalize; 3. prune.

- advantage:
  systematic transformation based on computation increment.
  know when optimized program may compute values not needed.
  (of course: no dynamic overhead, specialized efficient strategy.)

- what is not free:
  computation increment, but most are straightforward.
  equality reasoning in inc., but simple ones are powerful.

I. Determine Computation Increment

Identify how computation of expensive function f may proceed.

Motivation: maximize reuse $\longrightarrow$ minimize increment $\oplus$
extract minimal argument differences that cover all rec. calls.

Example:

$$\boxed{\begin{array}{lll} c(i,j) & \triangleq & \textbf{if } i=0 \vee j=0 \textbf{ then } 0 \\ & & \textbf{else if } x[i]=y[j] \textbf{ then } c(i-1,j-1)+1 \\ & & \textbf{else } \max(c(i,j-1),\, c(i-1,j)) \end{array}}$$

$\langle i-1, j-1\rangle$, $\underline{\langle i, j-1\rangle}$, $\underline{\langle i-1, j\rangle}$ $\quad\quad$ $\langle i,j\rangle = \langle i'-1, j'\rangle$

$\langle i', j'\rangle = \langle i+1, j\rangle$

Algorithm:

1. collect arguments of all recursive calls;
2. represent them in terms of formal parameters;
3. compute set of minimal differences;
4. remove from it any one that is covered by remaining ones.

II.1. Cache All Possibly Computed Values

Embed values of possibly computed function calls in return value.

Example:

$$\begin{array}{|lll|} \hline c(i,j) & \triangleq & \textbf{if } i=0 \vee j=0 \textbf{ then } 0 \\ & & \textbf{else if } x[i]=y[j] \textbf{ then } c(i-1,j-1)+1 \\ & & \textbf{else } \max(c(i,j-1),\, c(i-1,j)) \\ \hline \bar{c}(i,j) & \triangleq & \textbf{if } i=0 \ \vee j=0 \textbf{ then } <0> \\ & & \textbf{else let } v_1 = \bar{c}(i-1,j-1) \textbf{ in} \\ & & \quad \textbf{let } v_2 = \bar{c}(i,j-1) \textbf{ in} \\ & & \quad \textbf{let } v_3 = \bar{c}(i-1,j) \textbf{ in} \\ & & \quad \textbf{if } x[i]=y[j] \textbf{ then } <1st(v_1)+1,\ v_1,\ v_2,\ v_3> \\ & & \quad \textbf{else } < \max(1st(v_2), 1st(v_3)),\ v_1,\ v_2,\ v_3> \\ \hline \end{array}$$

Issue: aux. info: hoist out of conditions that dep. on global vars.
both branches may be executed to terminate correctly.

II.2. Incrementalize

Derive one incremental version for each function definition.

$$\bar{f}(x) = \bar{r} \qquad x \oplus y = x' \qquad \bar{f}(x') \quad \longrightarrow \quad \bar{f}'(x', r)$$

- maintain invariants: about cached results and input change. results incl. return value, intermediate results, and aux. info.
- use invariants: replace fun. calls by retrievals from cached res.

Example: $\quad i' = i+1, \quad j' = j, \quad \bar{r} = \bar{c}(i,j)$

$$\begin{array}{l}
\bar{c}(i,j) \triangleq \\
\textbf{if } i=0 \lor j=0 \textbf{ then } < 0 > \\
\textbf{else let } v_1 = \boxed{\bar{c}(i-1,j-1)} \textbf{ in} \\
\quad \textbf{let } v_2 = \boxed{\bar{c}(i,j-1)} \textbf{ in} \\
\quad \textbf{let } v_3 = \boxed{\bar{c}(i-1,j)} \textbf{ in} \\
\quad \textbf{if } x[i] = y[j] \textbf{ then} \\
\qquad < 1st(v_1)+1,\ v_1,\ v_2,\ v_3 > \\
\quad \textbf{else } < \max(1st(v_2), 1st(v_3)), \\
\qquad\quad v_1,\ v_2,\ v_3 >
\end{array}$$

$$\begin{array}{ll}
\bar{c}(i',j') \triangleq & \bar{c}'(i',j',\bar{r}) \\
\textbf{if } i'=0 \lor j'=0 \textbf{ then } < 0 > & \\
\textbf{else let } v_1 = \boxed{\bar{c}(i'-1,j'-1)} \textbf{ in} & \bar{c}(i,j-1) \\
\quad \textbf{let } v_2 = \boxed{\bar{c}(i',j'-1)} \textbf{ in} & \bar{c}'(i',j'-1,c(i,j-1)) \\
\quad \textbf{let } v_3 = \boxed{\bar{c}(i'-1,j')} \textbf{ in} & \bar{c}(i,j) \\
\quad \textbf{if } x[i'] = y[j'] \textbf{ then} & \\
\qquad < 1st(v_1)+1,\ v_1,\ v_2,\ v_3 > & \\
\quad \textbf{else } < \max(1st(v_2), 1st(v_3)), & \\
\qquad\quad v_1,\ v_2,\ v_3 > &
\end{array}$$

II.2. Incrementalize

Transform based on semantics of language constructs.

- exploit control structures — conditional
- exploit data structures — constructor (tuple, rec), array

Example: $i' = i+1, \ j' = j$
$\bar{r} = \bar{c}(i,j)$

$$\begin{array}{l}
\bar{c}(i,j) \triangleq \\
\textbf{if } i=0 \lor j=0 \textbf{ then } < 0 > \\
\textbf{else let } v_1 = \boxed{\bar{c}(i-1,j-1)} \textbf{ in} \\
\quad \textbf{let } v_2 = \boxed{\bar{c}(i,j-1)} \textbf{ in} \\
\quad \textbf{let } v_3 = \boxed{\bar{c}(i-1,j)} \textbf{ in} \\
\quad \textbf{if } x[i] = y[j] \textbf{ then} \\
\qquad < 1st(v_1)+1,\ v_1,\ v_2,\ v_3 > \\
\quad \textbf{else } < \max(1st(v_2), 1st(v_3)), \\
\qquad\quad v_1,\ v_2,\ v_3 >
\end{array}$$

$$\begin{array}{lll}
\bar{c}(i',j') \triangleq & \bar{c}'(i',j',\bar{r}) & \\
\textbf{if } i'=0 \lor j'=0 \textbf{ then } < 0 > & & \\
\textbf{else if } i'-1=0 \lor j'=0 \textbf{ then} & & \\
\quad \textbf{let } v_1 = \boxed{\bar{c}(i'-1,j'-1)} \textbf{ in} & < 0 > & \\
\quad \textbf{let } v_2 = \boxed{\bar{c}(i',j'-1)} \textbf{ in} & \bar{c}'(i',j'-1,< 0 >) & \\
\quad \textbf{let } v_3 = \boxed{\bar{c}(i'-1,j')} \textbf{ in} & < 0 > & \\
\quad \textbf{if } x[i'] = y[j'] \textbf{ then } < 1st(v_1)+1,\ v_1,\ v_2,\ v_3 > & & \\
\quad \textbf{else } < \max(1st(v_2), 1st(v_3)),\ v_1,\ v_2,\ v_3 > & & \\
\textbf{else let } v_1 = \boxed{\bar{c}(i'-1,j'-1)} \textbf{ in} & \bar{c}(i,j-1) & 3rd(\bar{r}) \\
\quad \textbf{let } v_2 = \boxed{\bar{c}(i',j'-1)} \textbf{ in} & \bar{c}'(i',j'-1,c(i,j-1)) & \\
\quad \textbf{let } v_3 = \boxed{\bar{c}(i'-1,j')} \textbf{ in} & \bar{c}(i,j) & \bar{r} \\
\quad \textbf{if } x[i'] = y[j'] \textbf{ then } < 1st(v_1)+1,\ v_1,\ v_2,\ v_3 > & & \\
\quad \textbf{else } < \max(1st(v_2), 1st(v_3)),\ v_1,\ v_2,\ v_3 > & &
\end{array}$$

II.3. Prune Unnecessary Values

Prune cached values not used in the incremental computation.

- precise dep. analysis of recursive data structures [Liu 98,LS 99]

Example:

$$\begin{array}{l}
\bar{c}(i',j') \triangleq \\
\textbf{if } i'=0 \lor j'=0 \textbf{ then } < 0 > \\
\textbf{else if } i'-1=0 \lor j'=0 \textbf{ then} \\
\quad \textbf{let } v_1 = < 0 > \textbf{ in} \\
\quad \textbf{let } v_2 = \bar{c}'(i',j'-1,< 0 >) \textbf{ in} \\
\quad \textbf{let } v_3 = < 0 > \textbf{ in} \\
\quad \textbf{if } x[i'] = y[j'] \textbf{ then } < 1st(v_1)+1,\ \underline{v_1},\ v_2,\ \underline{v_3} > \\
\quad \textbf{else } < \max(1st(v_2), 1st(v_3)),\ \underline{v_1},\ v_2,\ \underline{v_3} > \\
\textbf{else let } v_1 = 3rd(\bar{r}) \textbf{ in} \\
\quad \textbf{let } v_2 = \bar{c}'(i',j'-1,3rd(\bar{r})) \textbf{ in} \\
\quad \textbf{let } v_3 = \bar{r} \textbf{ in} \\
\quad \textbf{if } x[i'] = y[j'] \textbf{ then } < 1st(v_1)+1,\ \underline{v_1},\ v_2,\ \underline{v_3} > \\
\quad \textbf{else } < \max(1st(v_2), 1st(v_3)),\ \underline{v_1},\ v_2,\ \underline{v_3} >
\end{array}$$

$$\begin{array}{l}
\bar{c}(i,j) \triangleq \\
\textbf{if } i=0 \lor j=0 \textbf{ then } < 0 > \\
\textbf{else let } v_1 = \bar{c}(i-1,j-1) \textbf{ in} \\
\quad \textbf{let } v_2 = \bar{c}(i,j-1) \textbf{ in} \\
\quad \textbf{let } v_3 = \bar{c}(i-1,j) \textbf{ in} \\
\quad \textbf{if } x[i] = y[j] \textbf{ then} \\
\qquad < 1st(v_1)+1,\ \underline{v_1},\ v_2,\ \underline{v_3} > \\
\quad \textbf{else } < \max(1st(v_2), 1st(v_3)), \\
\qquad\quad \underline{v_1},\ v_2,\ \underline{v_3} >
\end{array}$$

obtain $\hat{c}'$ and $\hat{c}$!

III. Form Optimized Program

Use the incremental version to form an optimized program.

- call incremental version repeatedly to compute final result.

$$\begin{array}{ll}
f(x) \triangleq & 1st(\hat{f}(x)) \\
\hat{f}(x) \triangleq & \textbf{if } base_cond(x) \textbf{then } base_val(x) \\
& \textbf{else let } \hat{r} = \hat{f}(prev(x)) \textbf{ in } \hat{f}'(x, \hat{r})
\end{array}$$

Example:

$$\begin{array}{ll}
c(i,j) \triangleq & 1st(\hat{c}(i,j)) \\
& \\
\hat{c}(i,j) \triangleq & \textbf{if } i=0 \lor j=0 \textbf{ then } < 0 > \\
& \textbf{else let } \hat{r} = \hat{c}(i-1,j) \textbf{ in } \hat{c}'(i,j,\hat{r})
\end{array}$$

quadratic time, linear space.

Summary

- identify f, $\oplus$; incrementalize to obtain f'; form optimized f.
  exploit results of all function applications
  use static analysis of control structures and data structures.

- dynamic programming out of a general systematic method.
  the principle applies also to loop, arrays, ...
  save time, reduce space, result in simple code...

- not free:
  computation increment, but most are straightforward.
  equality reasoning, but simple ones are powerful.

More Examples

Examples	multi cache argu	aux info	original program's running time	optimized prog's running time
Fibonacci function			$O(2^n)$	$O(n)$
Fibolucci function, etc.			$O(2^n)$	$O(n)$
binomial coefficients			$O(2^n)$	$O(n*k)$
longest common subseq.		$\surd$	$O(2^{n+m})$	$O(n*m)$
mtx-chain multiplication	$\surd$		$O(n*3^n)$	$O(n^3)$
string editing distance			$O(3^{n+m})$	$O(n*m)$
dag path sequence		$\surd$	$O(2^n)$	$O(n^2)$
optimal polygon triang.	$\surd$		$O(n*3^n)$	$O(n^3)$
optimal bin. search trees	$\surd$		$O(n*3^n)$	$O(n^3)$
paragraph formatting	$\surd$		$O(n*2^n)$	$O(n^2)$
paragraph formatting 2	$\surd$		$O(n*2^n)$	$O(n*width)$
0-1 knapsack		$\surd$a	$O(2^n)$	$O(n*weight)$
CFG parsing	$\surd$	$\surd$a	$O(n*(2*size+1)^n)$	$O(n^3*size)$

Related Work

- algorithm texts [AHU 74, CLR 90] and many algorithms [...]:
  manual.

- memoization [Michie 68, Hughes 85, Pugh Tei 89, Abadi Lam Levy 96]:
  dynamic, cache strategy not both general and efficient.

- tabulation:
  - early general strategies [Burstall Darlington 77, Scherlis 81]
    survey [Bird 80], collection [Partsch 90]
  - patterns of redundant recursive calls [Cohen 83-TOPLAS]
    tupling [Pettorossi 84, Chin 93, Chin Khoo 93, Hu et al 97-ICFP]
    list introduction [Pettorossi Proietti 97-TC2]
    vector-based memo [Chin Hagiya 97-ICFP]
  - programming with invariants [Paige 86]
    grammar thinning [Webber 95-TOPLAS]
    incrementalization [Liu Tei 95-SCP&PEPM, Liu Stoller Tei 96-POPL]

[LST 98-TOPLAS] Static Caching for Incremental Computation

Conclusion

- dynamic programming via static incrementalization
  - simple and powerful
  - systematic and general

- on-going improvements:
  - combine techniques for more optimization.
    merge techniques for more language features.
  - cost analysis to characterize speedup analytically.
    precise and precise dependence analysis for recursive data.
    equality reasoning, with greedy property.
  - experimentation: measuring performance improvements.
    implementation: Cachet; powerful, convenient meta-tools.