Part IV. Advanced design and analysis techniques

Chapter 15. Dynamic programming

Chapter 16. Greedy algorithms

Chapter 15. Dynamic programming

Optimization problems: solutions, optimal solutions, optimal cost

Problems solvable via divide-and-conquer approaches

Issues in dynamic programming

Examples:

(1) matrix-chain multiplication

(2) longest common subsequence

Overlapping subproblems

dividing a problem into subproblems, e.g.,

$$F(n) = F(n-1) + F(n-2),\ F(1) = F(2) = 1.$$

A direct implementation of a recursive approach leads to exponential running time.

Instead, a one-dimensional table $T[1..n]$ to look up would help to reduce running time.

Then it is to compute $T[n]$, the last cell of the table.

The computation can be done by scanning/filling the table from left to right.

Steps for dynamic programming

(1) the structure of optimal solutions

(2) defining optimal cost recursively

(3) computing optimal cost

(4) constructing optimal solution

Matrix-chain multiplication

INPUT: given n matrices $A_1, \cdots, A_n$, where A_i has dimension $p_{i-1} \times p_i$

OUTPUT: a parenthesization by which the product $A_1 \times A_2 \times \cdots \times A_n$ uses the minimum number of scalar multiplications.

Note: The number of all possible parenthesizations $P(n) = \Sigma_{k=1}^{n-1} P(k)P(n-k)$, Catalan number.

A dynamic programming approach:

step 1: find the structure of a solution, I.e., Optimal solution in terms of optimal solutions to subproblems: *optimal substructure.*

(1) the the best parenthesization of $A_iA_{i+1}\cdots A_j$ must be

$$(A_i \cdots A_k)(A_{k+1} \cdots A_j)$$

for some $k, i \leq k < j$.

(2) The optimal cost of $A_iA_{i+1}\cdots A_j$ must be the smallest among optimal costs for

$$(A_i \cdots A_k)(A_{k+1} \cdots A_j)$$

$k = i, i+1, \cdots, j-1$.

(3) For each k, the optimal cost for the above is the optimal cost for $A_i \cdots A_k$ plus the optimal cost for $A_{k+1} \cdots A_j$ plus the cost for multiplying these two terms.

step 2: a recursive solution.

Define $m[i, j]$ to be the minimum number of scalar multiplications needed for $A_i A_{i+1} \cdots A_j$.

Then

$$m[i, j] = \min_{i \leq k < j} \{m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j\}$$

$m[i, j] = 0$ when $i = j$.

step 3: computing the optimal cost:

```
MATRIX-CHAIN-ORDER(p)
n <-- length[p] -1
for i <-- 1 to n
   do m[i, i] <-- 0
for L <-- 2 to n
   do for i <-- 1 to n - L + 1
       do j <-- i + L -1
           m[i, j] <-- infinite
           for k <-- i to j - 1
              do q <- m[i,k]+m[k+1,j]+p[i-1]p[k]p[j]
                  if q < m[i, j]
                     then m[i, j] <-- q
                          s[i, j] <-- k
return m and s
```

Analysis of time complexity?

Longest Common Subsequence LCS

ACCGGTCGAGTGCG

GTCGTTCGGAATGC

the longest common subsequence is

CGTCGATGC

fomally, let $X = \langle x_1, x_2, \cdots, x_m \rangle$ be a sequence

another sequence $Z = \langle z_1, z_2, \cdots, z_k \rangle$ is a *subsequence* of X if there are

$i_1 < i_2 < \cdots, < i_k$ indices of X such that $x_{i_j} = z_j$ for $j = 1, \cdots, k$.

Z is a *common subsequence* of X and Y if Z is a subsequence of X and Z is a subsequence of Y.

LCS problem:

Input: two sequences $X = \langle x_1, x_2, \cdots, x_m \rangle$ and $Y = \langle y_1, y_2, \cdots, y_n \rangle$.

Output: the longest common subsequence of X and Y.

algorithms ??

A dynamic programming approach:

step 1. optimal substructure

step 2. a recursive solution

step 3. computing the longest length of an LCS

step 4. constructing a longest common subsequence

step 1. optimal substructure

$X = \langle x_1, x_2, \cdots, x_m \rangle$

$Y = \langle y_1, y_2, \cdots, y_n \rangle$

(1) If $x_m = y_n$ then the LCS of X and Y is the LCS for $\langle x_1, x_2, \cdots, x_{m-1} \rangle$ and $\langle y_1, y_2, \cdots, y_{n-1} \rangle$ appended with x_m;

(2) If $x_m \neq y_n$, then the LCS of X and Y is either

the LCS for $\langle x_1, x_2, \cdots, x_{m-1} \rangle$ and Y or the LCS for X and $\langle y_1, y_2, \cdots, y_{n-1} \rangle$.

step 2. a recursive solution

Define $m[i, j]$ to be the length of a LCS for $\langle x_1, x_2, \cdots, x_i \rangle$ and $\langle y_1, y_2, \cdots, y_j \rangle$. Then

$m[i, 0] = 0$ and $m[0, j] = 0$;

when $x_i = y_j$, $m[i, j] = m[i-1, j-1] + 1$ and

when $x_i \neq y_j$, $m[i, j] = \max\{m[i-1, j], m[i, j-1]\}$.

step 3. computing the longest length of an LCS

```
LCS-LENGTH(X, Y)
m <-- length[X]
n <-- length[Y]
for i <--1 to m
   do m[i, 0] <-- 0
for j <-- 0 to n
   do m[0, j] <-- 0
for i <-- 1 to m
   do for j <-- 1 to n
        do if X[i] = Y[j]
             then m[i, j] <-- m[i-1,j-1] + 1
                  s[i, j] <-- "\"
             else if m[i-1, j] > m[i, j-1]
                    then m[i, j] <-- m[i-1, j]
                         s[i, j] <-- "|"
                    else m[i, j] <-- m[i, j-1]
                         s[i, j] <-- "-"
return m and s
```

step 4. constructing a longest common subsequence

```
PRINT-LCS(s, X, i, j)

if i = 0 or j=0
 then return
if s[i, j] = '\'
  then PRINT-LCS(s, X, i-1, j-1)
       print(X[i])
  else if s[i, j] = '|'
         then PRINT-LCS(s, X, i-1, j)
         else PRINT-LCS(s, X, i, j-1)
```

Running time?

Pairwise Sequence alignment

INPUT: $X, Y \in \{A, C, G, T\}^*$

OUTPUT: X', Y', where X' and Y' are of the same length r obtained from X and Y by inserting spaces '␣', and the score $\Sigma_{i=1}^{r} d(X'[i], Y'[i])$ is the maximum.

where *scoring matrix* $d_{5\times 5}$

$d(a, b) = 1$ if $a = b$ and neither is a '␣'

$d(a, b) = -1$ if $a \neq b$ and neither is a '␣'

$d(a, b) = -2$ if either is a '␣'.

The LCS problem is a special case of pairwise sequence alignment where $d(a, b) = 1$ if $a = b$; and 0 otherwise.

Dynamic programming for pairwise sequence alignment

step 1: problem analysis and finding recursive solutions

Consider aligning prefixes

$x_1, \ldots, x_i$

$y_1, \ldots, y_j$

(1) x_i is aligned to y_j, reducing the problem to aligning

$x_1, \ldots, x_{i-1}$

$y_1, \ldots, y_{j-1}$

(2) x_i is aligned to gap '_', reducing the problem to aligning

$x_1, \ldots, x_{i-1}$

$y_1, \ldots, y_j$

(3) y_j is aligned to gap '_', reducing the problem to

$x_1, \ldots, x_i$

$y_1, \ldots, y_{j-1}$

step 2: Define optimal cost function recursively

Define $S(i, j)$ to be the optimal score of the alignment between

$$x_1, \ldots, x_i \quad \text{and} \quad y_1, \ldots, y_j$$

Then $S(i, j)$ has the recurrences:

$$\begin{aligned} S(i,j) = \max\{ & S(i-1, j-1) + d(x_i, y_j), \\ & S(i-1, j) + d(x_i, '\_'), \\ & S(i, j-1) + d('\_', y_j) \ \} \end{aligned}$$

$$S(i, 0) = \sum_{k=1}^{i} d(x_i, '\_')$$

$$S(0, j) = \sum_{k=1}^{j} d('\_', y_j)$$

steps 3, 4 are similar to those for the LCS problem.

Multiple sequence alignment

INPUT: sequences $s_1, s_2, \ldots, s_k$
OUTPUT: an alignment A for these sequences such that
the SP score achieves the maximum

where SP, the *sum of pairs*, is $\sum_{i<j} C_{i,j}$, in which $C_{i,j}$ is the alignment score between s_i and s_j induced by the multiple alignment A.

Extending the dynamic programming for pairwise alignment to multiple alignment

Chapter 16 Greedy Algorithms

Dynamic programming is to *consider all possible choices and select the best.*

    always lead to the optimal solution

A greedy algorithm may *ignore some choices and select one that is locally the best.*

    a good greedy strategy may lead to the optimal solution

Activity-selection problem

INPUT: a set of activities, each with a start time and finish time

OUTPUT: a maximum size subset of mutually compatible activities.

A dynamic programming solution

step 1 analysis of problem

$S_{ij} = \{a_k \in S : f_i \leq s_k < f_k \leq s_j\}$

A_{ij} is such that

$$|A_{ij}| = \max_{a_k \in S_{ij}} \{|A_{ik} \cup \{a_k\} \cup A_{kj}|\}$$

step 2 define $c[i, j]$ to be the size of A_{ij}. Then recurrence

$c[i, j] = \max_{i<k<j}\{c[i, k] + c[k, j] + 1\}$.

Converting the dynamic programming solution to a greedy solution:

Theorem 16.1 Let S_{ij} be a non-empty set and $a_m \in S_{ij}$ with earliest finish time:

$$f_m = \min\{f_k : a_k \in S_{ij}\}$$

Then

(1) a_m is used in some maximum size subset of mutually compatible activities of S_{ij}

(2) the subproblem S_{im} is empty, so that choosing a_m leaves the subproblem S_{mj} as the only one that may be nonempty.

Significance of the theorem: it helps to reduce the number of subproblems to consider.

RECURSIVE-ACTIVITY-SELECTION(s, f, i, j)

1. m <-- i + 1
2. while m < j and $S_m < f_i$
3. m <-- m + 1
4. if $f_m < s_j$
5. then return $\{a_m\}$ U RECURSIVE-ACTIVITY-SELECTION(s, f, m, j)
6. else return 0

steps for the greedy strategy

1. determine the optimal substructure

2. develop a recursive solution

3. prove that at any stage of the recursion, one of the optimal choice is the greedy choice – it is safe to make that greedy choice

4. show that all but one of the subproblems induced by the greedy choice are empty.

example

Knapsack problems

input: n items $1, 2, \cdots, n$, each with size s_i and value v_i, a knapsack with size B,

output: a subset of items with total value maximized and total size $\leq B$.

0-1 Knapsack

fractional Knapsack

optimal substructure?

recursive solution?

greedy strategy?

(1) Does dynamic programming approach produce optimal solution for 0-1 Knapsack?

(2) Does dynamic programming approach runs in polynomial time on 0-1 knapsack?

(3) Does greedy approach produce optimal solution for 0-1 knapsack?

(4) Does greedy approach produce optimal solution for fractional knapsack?

(5) Does greedy approach runs in polynomial time on fractional knapsack?

Huffman Code

compressing data using binary bits

code: a compressing scheme

fixed-length code

variable-length code

character	a	b	c	d	e	f
frequency	.45	.13	.12	.16	.09	.05
fixed-length	000	001	010	011	100	101
var-length						

decoding process, code tree, prefix code

Finding an optimal prefix code (Huffman)

code tree

cost:

$$B(T) = \sum_{c \in C} f(c) d_T(c)$$

1. optimal substructure?
2. recursive solution?
3. greedy choice?
4. all but one subproblems induced by the greedy choice are empty?

5. development of a greedy algorithm

```
HUFFMAN(C, f)

n <-- |C|
Q <-- C
for i =1 to n-1
  do newnode(z)
     leftchild[z] <-- x <-- EXTRACT-MIN(Q)
     rightchild[z] <-- y <-- EXTRACT-MIN(Q)
     f(z) <-- f(x) + f(y)
     INSERT(Q, z)
return EXTRACT-MIN(Q)
```

Correctness of Huffman's algorithm

Lemma 16.2 (Greedy choice property)

Let C be an alphabet and f be the frequence function for characters in C. Let x and y be two characters in C having the lowest frequencies. Then there exists an optimal prefix code for C in which the codewords for x and y have the same length and differ in the last bit.

Proof:

Lemma 16.3 (Optimal substructure)

Let C be an alphabet and f be the frequence function for characters in C. Let x and y be two characters in C having the lowest frequencies. Let

$$C' = C - \{x, y\} \cup \{z\}$$

and the frequency function for C' is the same as f except that $f(z) = f(x) + f(y)$.

Let T' be any an optimal prefix code tree for C'. Then the tree T obtained from T' by replacing the leaf node z with an internal node having x and y as children, is an optimal prefix code tree for C.

Proof:

Theorem 16.4 Huffman's algorithm produces an optimal prefix code.