Dynamic programming techniques

Topics

- Basics of DP
- Matrix-chain Multiplication
- Longest Common subsequence
- All-pairs Shortest paths

Further Reading

Chapter 6

Textbook

Dynamic programming

- **Solves problems by combining the solutions to subproblems**

- **DP is applicable when subproblems are not independent**

  Subproblems share subsubproblems
  In such cases a simple
  Divide and Conquer strategy solves common subsubproblems.

- **In DP every subproblem is solved just once and the solution is saved in a table for future reference (avoids re-computation).**

- **DP is typically applied to optimization problems**

- **A given problem may have many solutions, DP chooses the optimal solution.**

Four stages of Dynamic Programming

- Characterize the structure of an optimal solution
- Recursively define the value of an optimal solution
- Compute the value of an optimal solution in a bottom-up fashion
- Construct an optimal solution from computed results

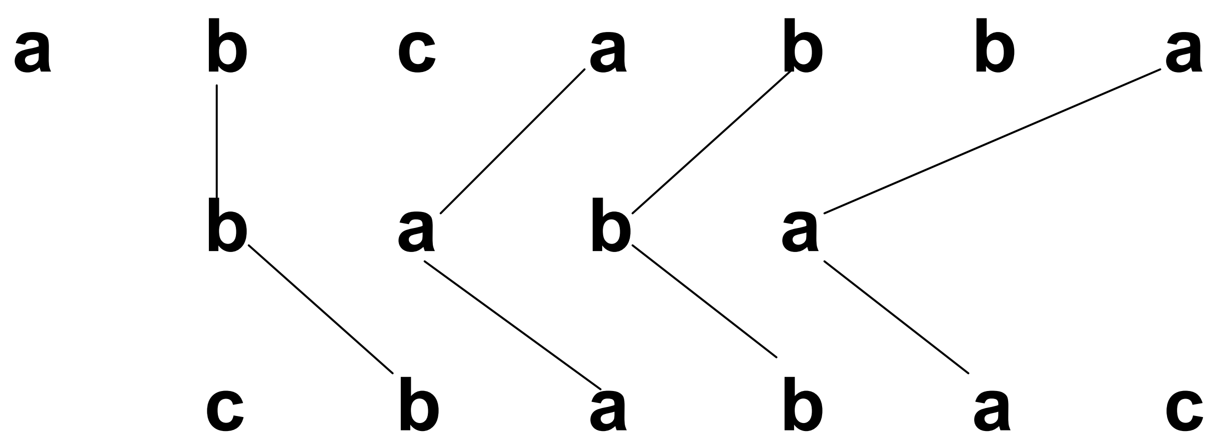
Longest common subsequence

A subsequence is formed from a list by deleting zero or more elements (the remaining elements are in order)

A common subsequence of two lists is a subsequence of both.
The longest common subsequence (LCS) of two lists is the longest among the common subsequences of the two lists.

Example:

abcabba and cbabac are two sequences
baba is a subsequence of both


a b c a b b a
b a b a
c b a b a c

To find the length of an LCS of lists x and y, we need to find the lengths of the LCSs of all pairs of prefixes.

- **a prefix is an initial sublist of a list**

If $x = (a_1, a_2, a_3, \ldots, a_m)$ and
$y = (b_1, b_2, b_3, \ldots, b_n)$
$0 \le i \le m$ and $0 \le j \le n$

Consider an LCS of the prefix $(a_1, a_2, a_3, \ldots, a_i)$ from x and of the prefix $(b_1, b_2, b_3, \ldots, b_j)$ from y.

If i or j = 0 then one of the prefixes is ε and the only possible common subsequence between x and y is ε and the length of the LCS is zero.

L(i,j) is the length of the LCS of $(a_1,a_2,a_3, . . ., a_i)$ and $(b_1,b_2,b_3, . . ., b_j)$.

BASIS: If $i+j = 0$, then both *i* and *j* are zero and so the LCS is ε.

INDUCTION: Consider *i* and *j*, and suppose we have already computed L(*g,h*) for any *g* and *h* such that

$$g+h < i+j.$$

1. If either *i* or *j* is 0 then $L(i,j) = 0$.
2. If $i>0$ and $j>0$, and $a_i \neq b_j$ then
   $L(i,j) = \max(L(i,j-1),L(i-1,j))$.
3. If $i > 0$ and $j > 0$, and $a_i = b_j$ then $L(i,j) = L(i-1,j-1)+1$.

1. If either i or j is 0 then $L(i,j) = 0$.
2. If $i>0$ and $j>0$, and $a_i \neq b_j$ then
   $L(i,j) = \max(L(i,j-1), L(i-1,j))$.
3. If $i>0$ and $j>0$, and $a_i = b_j$ then $L(i,j) = L(i-1,j-1)+1$.

b	0	1	2	2	3
a	0	1	1	2	3
c	0	1	1	2	2
a	0	1	1	1	1
ε	0	0	0	0	0
	ε	a	b	c	a

Procedure LCS(*x*,*y*)
Input : The lists *x* and *y*
Output : The longest common subsequence and its length

```
for j ← 0 to n do
        L[0,j] ← 0;
for  i ← 1 to m do
        L[i,0] ←0;
        for j ← 1 to n do
                if a[i] ≠ b[j] then
                        L[i,j] ← max {L[i-1,j],L[i,j-1]};
                else
                        L[i,j] ← 1+L[i-1,j-1];
```

Example:

Consider, lists x = abcabba and y = cbabac

c	6	0	1	2	3	3	3	3	4
a	5	0	1	2	2	3	3	3	4
b	4	0	1	2	2	2	3	3	3
a	3	0	1	1	1	2	2	2	3
b	2	0	0	1	1	1	2	2	2
c	1	0	0	0	1	1	1	1	1
0	0	0	0	0	0	0	0	0	0
		0	a	b	c	a	b	b	a

Consider another example

abaacbacab and bacabbcaba LCS : bacacab

b	0	1	2	3	4	5	5	5	6	7	7
a	0	1	2	3	4	4	4	5	6	6	6
c	0	1	2	3	4	4	4	5	5	5	6
a	0	1	2	3	4	4	4	4	5	5	6
b	0	1	2	3	3	4	4	4	4	5	5
c	0	1	2	3	3	3	3	4	4	4	4
a	0	1	2	2	3	3	3	3	3	3	4
a	0	1	2	2	2	2	2	2	3	3	3
b	0	1	1	1	1	2	2	2	2	2	2
a	0	0	1	1	1	1	1	1	1	1	1`
0	0	0	0	0	0	0	0	0	0	0	0
	0	b	a	c	a	b	b	c	a	b	a

- Give a dynamic-programming solution to the 0-1 knapsack problem that runs in $O(nW)$ time, where n is the number of items and W is the maximum weight of items that the thief can put in his knapsack.

The weight is measured in Kgs (say). The maximum weight is an integer. The given items are 1..*n*

Let *S* be the optimal solution for *W* and *i* be the highest numbered item in *S*. $S' = S - \{i\}$ is an optimal solution for $(W-w_i)$ Kilos and items 1.. *i*-1.
The value of the solution in *S* is the value v_i of item *i* plus the value of the solution *S*'.
Let $c[i,w]$ be the value of the solution for items 1..*i* and maximum weight *w*.

$$C[i,w] = \begin{cases} 0 & \text{if } i=0 \text{ or } w=0. \\ c[i-1,w] & \text{if } w_i > w \\ \max(v_i + c[i-1, w-w_i], c[i-1,w]) & \text{if } i>0 \text{ and } w \geq w_i \end{cases}$$

The value of the solution for *i* items either includes item *i*, in which case it is v_i plus a Subproblem solution for *i*-1 items and the weight excluding w_i or doesn't include the item *i*.

- Give a dynamic-programming solution to the 0-1 knapsack problem that runs in $O(nW)$ time, where n is the number of items and W is the maximum weight of items that the thief can put in his knapsack.

Inputs : W, n, $v = \langle v_1, v_2, \ldots, v_n \rangle$ and $w = \langle w_1, w_2, \ldots, w_n \rangle$

The table is $c[0..n, 0..W]$ – each entry is referred to as $c[i,j]$

The first row entries are filled first and then the second row entries are computed and so on (very similar to the LCS solution).

At the end $c[n,W]$ contains the maximum value.
Trace the items which are part of the solution from $c[n,W]$.
If $c[i,w] = c[i-1,w]$ then i is not part of the solution, go to $c[i-1,w]$ and trace back
If $c[i,w] \neq c[i-1,w]$ then i is part of the solution, trace with $c[i-1,w-w_i]$.

Item	Weight	Value
1	2	12
2	1	10
3	3	20
4	2	15

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

Matrix-chain Multiplication

Consider the matrix multiplication procedure

```
MATRIX_MULTIPLY(A,B)
if columns[A] ≠ rows[B]
    then error "incompatible dimensions”
    else for i ← 1 to rows[A]
            do for j ←1 to columns[B]
                do C[i,j] ←0;
                    for k ← 1 to columns [A]
                        do C[i,j] ← C[i,j]+A[i,k]*B[k,j];
        return C
```

The time to compute a matrix product is dominated by the number of scalar multiplications in line 7.

If matrix A is of size ($p \times q$) and B is of size ($q \times r$), then the time to compute the product matrix is given by *pqr*.

Consider three matrices A1, A2, and A3 whose dimensions are respectively

(10×100), (100×5), (5×50).

Now there are two ways to parenthesize these multiplications

I $((A_1 \times A_2) \times A_3)$

II $(A_1 \times (A_2 \times A_3))$

First Parenthesization

Product $A_1 \times A_2$ requires $10 \times 100 \times 5 = 5000$ scalar multiplications
$A_1 \times A_2$ is a (10×5) matrix
$(A_1 \times A_2) \times A_3$ requires $10 \times 5 \times 50 = 2500$ scalar multiplications.
Total : 7,500 multiplications

$$\begin{bmatrix} 10\times100 \\ \\ A_1 \end{bmatrix} \times \begin{bmatrix} 100\times5 \\ \\ A_2 \end{bmatrix} = \begin{bmatrix} 10\times5 \\ \\ A_1\times A_2 \end{bmatrix} \qquad A_1$$

$$\begin{bmatrix} 10\times5 \\ \\ A_1\times A_2 \end{bmatrix} \times \begin{bmatrix} 5\times50 \\ \\ A_3 \end{bmatrix} = \begin{bmatrix} 10\times50 \\ \\ (A_1\times A_2) \\ \times A_3 \end{bmatrix}$$

Second Parenthesization

Product $A_2 \times A_3$ requires $100 \times 5 \times 50$ = 25,000 scalar multiplications
$A_2 \times A_3$ is a (100×50) matrix
$A_1 \times (A_2 \times A_3)$ requires $10 \times 100 \times 50$ = 50,000 scalar multiplications
Total : 75,000 multiplications.

$$\begin{bmatrix} 100 \times 5 \\ \\ A_2 \end{bmatrix} \times \begin{bmatrix} 5 \times 50 \\ \\ A_3 \end{bmatrix} = \begin{bmatrix} 100 \times 50 \\ \\ A_2 \times A_3 \end{bmatrix}$$

$$\begin{bmatrix} 10 \times 100 \\ \\ A_1 \end{bmatrix} \times \begin{bmatrix} 100 \times 50 \\ \\ A_2 \times A_3 \end{bmatrix} = \begin{bmatrix} 10 \times 50 \\ \\ A_1 \times \\ (A_2 \times A_3) \end{bmatrix}$$

The first parenthesization is 10 times faster than the second one!!

How to pick the best parenthesization ?

The matrix-chain matrix multiplication

Given a chain $(A_1, A_2, \ldots, A_n)$ of n matrices, where for i = 1,2, …, n matrix A_i has dimension $p_{i-1} \times p_i$, fully parenthesize the product $A_1A_2 \ldots A_n$ in a way that minimizes the number of scalar multiplications.

The order in which these matrices are multiplied together can have a significant effect on the total number of operations required to evaluate the product.

An optimal solution to an instance of a matrix--chain multiplication problem contains within it optimal solutions to the subproblem instances.

Let, *P(n)* : The number of alternative parenthesizations of a sequence of *n* matrices

We can split a sequence of n matrices between kth and (k+1)st matrices for any k = 1, 2, …, n-1 and we can then parenthesize the two resulting subsequences independently,

$$P(n) = \begin{cases} 1 \ \ if \ \ n = 1 \\ \sum_{k=1}^{n-1} P(k) \cdot P(n-k) \ \ if \ \ n \geq 2 \end{cases}$$

This is an exponential in *n*

Consider $A_1 \times A_2 \times A_3 \times A_4$

if k =1, then

$A_1 \times (A_2 \times (A_3 \times A_4))$ or

$A_1 \times ((A_2 \times A_3) \times A_4)$

if k =2 then

$(A_1 \times A_2) \times (A_3 \times A_4)$

if k =3 then

$((A_1 \times A_2) \times A_3) \times A_4$

or $(A_1 \times (A_2 \times A_3)) \times A_4$

Structure of the Optimal Parenthesization

$A_{i..j} = A_i \times A_{i+1} \times \ldots \times A_j$

An optimal parenthesization splits the product

$A_{i..j} = (A_i \times A_{i+1} \times \ldots \times A_k) \times (A_{k+1} \times A_{k+2} \times \ldots \times A_j)$ for $1 \leq k < n$

The total cost of computing $A_{i..j}$

= cost of computing $(A_i \times A_{i+1} \times \ldots \times A_k)$

+ cost of computing $(A_{k+1} \times A_{k+2} \times \ldots \times A_j)$

+ cost of multiplying the matrices $A_{i..k}$ and $A_{k+1..j}$.

$A_{i...k}$ must also be optimal if we want $A_{i..j}$ to be optimal. If $A_{i..k}$ is not optimal then $A_{i..j}$ is not optimal. Similarly $A_{k+1..j}$ must also be optimal.

Recursive Solution

We'll define the value of an optimal solution recursively in terms of the optimal solutions to subproblems.

$m[i,j]$ = minimum number of scalar multiplications needed to compute the matrix $A_{i..j}$
$m[1,n]$ = minimum number of scalar multiplications needed to compute the matrix $A_{1..n}$.

If $i = j$; the chain consists of just one matrix
$A_{i..i} = A_i$ - no scalar multiplications
$m[i,i] = 0$ for $i = 1, 2, \ldots, n$.

$m[i,j]$ = minimum cost of computing the subproducts $A_{i..k}$ and $A_{k+1..j}$ + cost of multiplying these two matrices

Multiplying $A_{i..k}$ and $A_{k+1....j}$ takes $p_{i-1}p_k\, p_j$ scalar multiplications
$m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_k\, p_j$ for $i \leq k < j$

The optimal parenthesization must use one of these values for *k*, we need to check them all to find the best solution.

Therefore,

$$m[i,j] = \begin{cases} 0 \quad if \quad i = j \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j]\} + p_{i-1} p_k p_j \end{cases}$$

Let *s*[*i,j*] be the value of *k* at which we can split the product $A_i \times A_{i+1} \times \ldots \times A_j$ to obtain the optimal parenthesization.

$s[i,j]$ equals a value of *k* such that

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

Procedure Matrix_Chain_Order (p)

Input: sequence $(p_0, p_1, \ldots p_n)$

Output : an auxiliary table m[1..n,1..n] with m[i,j] costs and another auxiliary table s[1..n,1..n] with records of index k which achieves optimal cost in computing m[i,j]

```
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]←∞;
            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
```

Consider Four Matrices

A1 : 10 × 20 **A2 : 20×50**

A3: 50×1 **A4 : 1 ×100**

MIN[(10,000+500),(1000+200)

j↓ i→	1	2	3	4
1	0	--	--	--
2	10,000	0	--	--
3	1200	1000	0	--
4	2200	3000	5000	0

Consider $A_1 \times A_2 \times A_3 \times A_4$

if k =1, then

$A_1 \times (A_2 \times (A_3 \times A_4))$

$A_1 \times ((A_2 \times A_3) \times A_4)$

if k =2 then

$(A_1 \times A_2) \times (A_3 \times A_4)$

if k =3 then

$((A_1 \times A_2) \times A_3) \times A_4$

10 ×50

and $(A_1 \times (A_2 \times A_3)) \times A_4$

20 ×1

$$m[i,j] = m[i,k] + m[k+1,j] + p_{i-1} p_k p_j \text{ for } i \le k < j$$

Consider, A1 (30×35)A2 (35×15)A3 (15×5), A4(5×10), A5(10×20), A6(20×25)

m

j↓/i→	1	2	3	4	5	6
1	0	--	--	--	--	--
2	15,750	0	--	--	--	--
3	7,875	2,625	0	--	--	--
4	9,375	4,375	750	0	--	--
5	11,875	7,125	2,500	1,000	0	--
6	15,125	10,500	5,375	3,500	5,000	0

s

j↓/i→	1	2	3	4	5
2	1	-	-	-	-
3	1	2	-	-	-
4	3	3	3	-	-
5	3	3	3	4	-
6	3	3	3	5	5

(A1..A3)× (A4..A6)

(A1 × (A2 × A3)) ×

((A4 × A5) × A6)

Complexity? With and without DP?

- T(1) ≥ 1
- T(n) ≥ 1 + $\sum_{k=1}^{n-1} T(k) + T(n-k) + 1) \; for \;\; n \geq 1$
- T(n) ≥
- Exponential in *n*

$$2\sum_{i=1}^{n-1} T(i) + n$$

Consider the problem of neatly printing a paragraph on a printer. The input text is a sequence of *n* words of length $l_1, l_2, \ldots, l_n$, measured in input characters. We want to print this paragraph neatly on a number of lines that hold a maximum of *M* characters each. Our criterion of "neatness" is as follows. If a given line contains words *i* through *j* and leave exactly one space between words, the number of extra space characters at the end of the line is

$$M - j + i - \sum_{k=i}^{j} l_k$$

We wish to minimize the sum, over all the lines except the last of the extra space characters at the ends of lines. Give a dynamic programming algorithm to print a paragraph of *n* words neatly on a printer. Analyze the running time and space requirements of your algorithm.

Hints

- Assume that no word is longer than a line
- Determine the cost of a line containing words *i* through *j* (cost is the number of free spaces)
- We want to minimize the sum of line costs over all lines in the paragraph.
- Try to represent the above by a recursive expression

- Assume that no word has more characters than that can be fitted in a line $l_i \leq M$ for all i
- We use DP for the following reasons,
  - There are a number of repeated problems
  - Solutions have optimal substructure
    - That is, if we place words 1..k on line 1, then the placement of words k+1 ..n must be optimal, else we have to improve the solution

- We denote the space at the end of a line that contains words i through j by,
  - **Space [*i,j*] = $M-j+i-\Sigma l_k$ ($k = i$ through j)**
- *Let*
  - $c_l[i, j]$ = cost of including a line containing words i through j in the sum S we want to minimize
  - c[j] = cost of an optimal arrangement of words i through j
- When the words don't fit on a line, such sum should not be part of S
  - Therefore we assume that $c_l[i, j] = \infty$ when *space*[i, j] < 0
  - For the last line when j =n, *space*[i, j] =0, and therefore, $c_l[i, j]$ =0
  - For all other cases $c_l[i, j]$ = *space*[i, j]

- Therefore we assume that $c_l[i, j] = \infty$ when $space[i, j] < 0$
- For the last line when j =n, $space[i, j]$ =0, and therefore, $c_l[i, j]$ =0
- For all other cases $c_l[i, j] = space[i, j]$

- The problem is to minimize S (the sum of all c_l) over all lines of the paragraph
- Cost of optimal arrangement = c[n]
- C[n] is defined recursively as follows,
  - c[0] =0
  - $c[j] = \min_{1 \le i \le j} c[i-1+c_l[i,j]$
- To arrange the first j words on lines, pick some i such that words i..j will be on the last line.
- Cost of the whole arrangement is given by
  - Line cost for that line containing words *i..j* PLUS
  - Cost of an optimal arrangement of the first i-1 words on earlier lines
- To find i such that the cost is minimized

- All choices will fit in any line because of the assumption that no word is longer than a line

c	*c[0]*	*c[1]*	*c[2]*				*c[n]*
p	*0*	*p[1]*					*p[n]*

- When *c[j]* is computed, if *c[j]* is based on the value of *c[k],* set *p[j]* = *k*
- When *c[n]* is computed, we trace the pointers to see where to break the lines.
- The last line starts at word *p[n]*+1, the line preceding that will start at *p[p[n]]*+1.
- The p table entries point to where each c value came from (the corresponding i)
- Space = $\Theta(n)$
- Time = $\Theta(n*M)$

- A ski rental agency has *m* pairs of skis, where the height of the *i*th pair of skis is s_i. There are *n* skiers who wish to rent skis, where the height of the *i*th skier is h_i. Ideally, each skier should obtain a pair of skis whose height matches with his own height as closely as possible. Design an efficient algorithm to assign skis so that the sum of the absolute differences of the heights of each skier and his/her skis is minimized.