CS2 Guest Lecture - Guha's Class

Tuesday, April 16, 2024 2:27 PM

Dynamic Programming - Matrix chain Multiplication

The Problem:

Input of n matrices (sequence): $\langle A_1, A_2, \ldots, A_n \rangle$

Output is a mult. setup $A_1 * A_2 * \ldots * A_n$

Ex. $n=4$ $A_1 * A_2 * A_3 * A_4$

Possible Answers

1) $(((A_1 A_2) A_3) A_4)$

2) $((A_1 A_2)(A_3 A_4))$

3) $((A_1 (A_2 A_3)) A_4)$

... there's more

? Which one requires the least amount of scalar mult.?

Derive the Dynamic Programming Solution

Step 1) Optimize Parenthesization $A_i \ldots A_j$ which splits the product $A_i \ldots A_k$ and $A_{k+1} \ldots A_j$

Examine ALL candidates of k. $k = i, i+1, i+2, \ldots, j-1$

Step 2) Recursively Define the optimal solution

m[i][j]: stores the value of minimum # of scalar mult. for $A_i A_j$

The call: $m[i][j] = m[i][k] + m[k+1][j] + p_{i-1} p_k p_j$

Step 3) Apply DP (memoization or tabulation)

A[1][1]
A[2][2]
⋮
A[5][5]

Step 4) Construct the Solution

Example

p = {5, 10, 4, 15, 2, 20}

Example

$P = \langle 5, 10, 4, 15, 2, 20 \rangle$

P =	5	10	4	15	2	20
	0	1	2	3	4	5

$n = 5$ $\quad A_1$ 5×10 $\quad A_2$ 10×4 $\quad A_3$ 4×15 $\quad A_4$ 15×2 $\quad A_5$ 2×20

m

	1	2	3	4	5
1	0	200	500	300	500
2		0	600	200	600
3			0	120	280
4				0	600
5					0

s

	1	2	3	4	5
1		1	2	1	4
2			2	2	4
3				3	4
4					4
5					

$l = 2$

$A_1 A_2$ $\quad m[1][2] \overset{k=1}{=} m[1][1] + m[2][2] + P[0] * P[1] * P[2] = 200$

$A_2 A_3$ $\quad m[2][3] \overset{k=2}{=} m[2][2] + m[3][3] + P[1] * P[2] * P[3] = 600$

$A_3 A_4$ $\quad m[3][4] \overset{k=3}{=} m[3][3] + m[4][4] + P[2] * P[3] * P[4] = 120$

$A_4 A_5$ $\quad m[4][5] \overset{k=4}{=} m[4][4] + m[5][5] + P[3] * P[4] * P[5] = 600$

$l = 3$

$A_1 A_2 A_3$ $\quad m[1][3]$

$k=1$: $m[1][1] + m[2][3] + P[0] * P[1] * P[3] = 1350$

$k=2$: $m[1][2] + m[3][3] + P[0] * P[2] * P[3] = \boxed{500}$

$A_2 A_3 A_4$ $\quad m[2][4]$

$k=2$: $m[2][2] + m[3][4] + P[1] * P[2] * P[4] = \boxed{200}$

$k=3$: $m[2][3] + m[4][4] + P[1] * P[3] * P[4] = 900$

$A_3 A_4 A_5$ $\quad m[3][5]$

$k=3$: $m[3][3] + m[4][5] + P[2] * P[3] * P[5] = 1800$

$k=4$: $m[3][4] + m[5][5] + P[2] * P[4] * P[5] = \boxed{280}$

$l = 4$

$\ell = 4$

$A_1 A_2 A_3 A_4$ $m[1][4]$

- $k=1$: $m[1][1] + m[2][4] + p[0] * p[1] * p[4] = \boxed{300}$
- $k=2$: $m[1][2] + m[3][4] + p[0] * p[2] * p[4] = 360$
- $k=3$: $m[1][3] + m[4][4] + p[0] * p[3] * p[4] = 650$

$A_2 A_3 A_4 A_5$ $m[2][5]$

- $k=2$: $m[2][2] + m[3][5] + p[1] * p[2] * p[5] = 1080$
- $k=3$: $m[2][3] + m[4][5] + p[1] * p[3] * p[5] = 4200$
- $k=4$: $m[2][4] + m[5][5] + p[1] * p[4] * p[5] = \boxed{600}$

$\ell = 5$

$A_1 A_2 A_3 A_4 A_5$ $m[1][5]$

- $k=1$: $m[1][1] + m[2][5] + p[0] * p[1] * p[5] = 1600$
- $k=2$: $m[1][2] + m[3][5] + p[0] * p[2] * p[5] = 880$
- $k=3$: $m[1][3] + m[4][5] + p[0] * p[3] * p[5] = 2600$
- $k=4$: $m[1][4] + m[5][5] + p[0] * p[4] * p[5] = \boxed{500}$

Edit Distance Problem

Given an initial string *s*, and a target string *t*, what is the minimum number of operations that must be applied to *s* to turn it into *t*. The list of valid changes are:

1. Inserting a character
2. Deleting a character
3. Replacing a character

Input: Two Strings
Output: The minimum number of operations needed to be applied.

Understanding the Subproblems (Recursion Perspective)

What if we observed the characters from sort of the direction (left or right)?

Let's say we start at the right end (could do this from the left too).

Two Possibilities

- Match

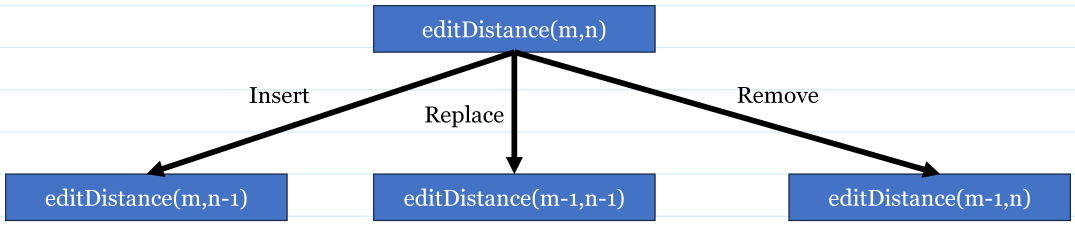
- No Match

If the characters match, then there is no need perform any of the three operations.

If the characters don't match, then we need to perform one of the three operations with the intention of performing the least amount operations.

Let's assume *s* is length *m* and *t* is length *n*.

The goal is to figure out the minimum number operations needed to convert *s* into *t*.



Insert: If we were to insert a character into string *s*, this comes with the assumption that our string *s* input is smaller than string *t* and we want that character to eventually match the respective character in string *t*. Since we insert a character in *s* to match a character in *t*, we can subtract one of the characters it matches in *t*, resulting in the traversal of n – 1.

Replace: If we were to replace a character into string *s*, this comes with the assumption that our string *s* input is the same size as string *t* and we want that character to match the respective character in string *t* at the same position. Since we are replacing a character in *s* to match a character in *t* at the same position, we can subtract the size of both *s* and *t*, resulting in the traversal of m – 1 and n – 1.

Remove: If we were to remove a character from string *s*, this comes with the assumption that our string *s* input is bigger than string *t* and there is no character it will correspond to with a respective character in string *t*. Since we removed a character in *s*, we can subtract the size of *s*, resulting in the traversal of m – 1.

Algorithm 1 EditDistance (String s, String t, int m, int n)

```
1: if m == 0 then
2:     return n
3: end if
4: if n == 0 then
5:     return m
6: end if
7: if s[m − 1] == t[n − 1] then
8:     return 0 + EditDistance(s, t, m − 1, n − 1)
9: end if
10: return 1 + min(EditDistance(s, t, m, n − 1), EditDistance(s, t, m − 1, n),
    EditDistance(s, t, m − 1, n − 1))
```

Apply Memoization

Algorithm 2 EditDistanceMemo (String s, String t, int m, int n, int[][] memo)

```
1: if m == 0 then
2:     return n
3: end if
4: if n == 0 then
5:     return m
6: end if
7: if memo[m][n] != −1 then
8:     return memo[m][n]
9: end if
10: if s[m − 1] == t[n − 1] then
11:     memo[m][n] = EditDistanceMemo(s, t, m − 1, n − 1, memo)
12:     return return memo[m][n]
13: else
14:     insert = EditDistanceMemo(s, t, m, n − 1, memo)
15:     replace = EditDistanceMemo(s, t, m − 1, n − 1, memo)
16:     remove = EditDistanceMemo(s, t, m − 1, n, memo)
17:     memo[m][n] = 1 + min(insert, replace, remove)
18:     return memo[m][n]
19: end if
```

Apply Tabulation

Algorithm 3 EditDistanceTab (String s, String t, int m, int n)

```
1: int dp[m + 1][n + 1]
2: for i = 1 to m do
3:     for j = 1 to n do
4:         if i == 0 then
5:             dp[i][j] = j
6:         else if j == 0 then
7:             dp[i][j] = i
8:         else if s[i − 1] == t[j − 1] then
9:             dp[i][j] = dp[i − 1][j − 1]
10:        else
11:            dp[i][j] = 1 + min(dp[i][j − 1], dp[i − 1][j], dp[i − 1][j − 1])
12:        end if
13:    end for
14: end for
15: return dp[m][n]
```