Design Techniques Part-2

Design Technique Part-2

Content :

DYNAMIC PROGRAMMING APPROACH

THE STRUCTURE OF AN OPTIMAL PARENTHESIZATION

Let us adopt the notation $A_{i..j}$ for the matrix that results from evaluating the product A_i A_{i+1} … A_j. It is the product A_1 A_2 … A_n. Splits the product between A_k and A_{k+1} for some integer k in the range $1 \leq k \leq n$ i.e. for few value of k, we first compute the matrices $A_{1..k}$ and $A_{k+1..n}$ and then multiply them together to produce the final product $A_{1..n}$. The cost of this is computing the matrix $A_{1..k}$ + the cost of computing $A_{k+1..n}$ + cost of multiplying them together.

Let m[I, j] be the minimized number of scalar multiplications needed to compute the matrix $A_{i..j}$, the cost of a cheapest way to compute $A_{1..n}$ would thus be m[1..n].

We can define m[i..j] recursively as follows:

If i = j the chain consists of just one matrix $A_{i..j} = A_i$ so no scalar multiplication are necessary to compute the product. Thus m[i, j] = 0 for i = 1, 2, 3, …, n.

To compute m[i, j], when i < j. Let us assume that the optimal parenthesization splits the product A_i A_{i+1} … A_j between A_k and A_{k+1} where $i \leq k \leq j$. then m[i, j] is equal to the minimum cost for computing the subproducts $A_{i..k}$ and $A_{k+1..j}$ + cost of multiplying them together. Since computing the matrix product $A_{i..k}$ and $A_{k+1..j}$ takes p_{i-1} p_k p_j scalar multiplications, we obtain.

$$M[i, j] = m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j$$

There are only (j-1) possible values for ‘k’ namely k = i, i + 1, …. j-1. It use one of these values for ‘k’, we need only check them all to find the best. So the minimum cost of parenthesizing the product A_i A_{i+1} … A_j becomes.

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

To construct an optimal solution, let us defined s[i, j] to be the value of ‘k’ at which we can split the product A_i A_{i+1}… A_j to obtain an optimal parenthesization i.e. s[i, j] = k such that

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

Example: We are given the sequence [4, 10, 3, 12, 20, 7]. The matrices have sizes 4×10, 10×3, 3×12, 12×20, 20×7. We need to compute M[i, j], 0 ≤ i, j ≤ 5. We know M[i, j] = 0 for all i.

1	2	3	4	5	
0					1
	0				2
		0			3
			0		4
				0	5

We proceed, working away from the diagonal. We compute the optimal solution for products of 2 matrices.

1	2	3	4	5	
0	120				1
	0	360			2
		0	720		3
			0	1680	4
				0	5

Now products of 3 matrices.

$$M[1,3]=\min\begin{cases} M[1,2]+M[3,3]+p_0p_2p_3 = 120+0+4\cdot 3\cdot 12 = 264 \\ M[1,1]+M[2,3]+p_0p_1p_3 = 0+360+4\cdot 10\cdot 12 = 840 \end{cases} = \mathbf{264}$$

$$M[2,4]=\min\begin{cases} M[2,3]+M[4,4]+p_1p_3p_4 = 360+0+10\cdot 12\cdot 20 = 2760 \\ M[2,2]+M[3,4]+p_1p_2p_4 = 0+720+10\cdot 3\cdot 20 = 1320 \end{cases} = \mathbf{1320}$$

$$M[3,5]=\min\begin{cases} M[3,4]+M[5,5]+p_2p_4p_5 = 720+0+3\cdot 20\cdot 7 = 1140 \\ M[3,3]+M[4,5]+p_2p_3p_5 = 0+1680+3\cdot 12\cdot 7 = 1932 \end{cases} = \mathbf{1140}$$

1	2	3	4	5	
0	120				1
	0	360			2
		0	720		3
			0	1680	4
				0	5

$\Rightarrow$

1	2	3	4	5	
0	120	264			1
	0	360	1320		2
		0	720	1140	3
			0	1680	4
				0	5

Now products of 4 matrics.

$$M[1,4]=\min\begin{cases} M[1,3]+M[4,4]+p_0p_3p_4 = 264+0+4\cdot 12\cdot 20 = 1224 \\ M[1,2]+M[3,4]+p_0p_2p_4 = 120+720+4\cdot 3\cdot 20 = 1080 \\ M[1,1]+M[2,4]+p_0p_1p_4 = 0+1320+4\cdot 10\cdot 20 = 2120 \end{cases} = \mathbf{1080}$$

$$M[2,5]=\min\begin{cases} M[2,4]+M[5,5]+p_1p_4p_5 = 1320+0+10\cdot 20\cdot 7 = 2720 \\ M[2,3]+M[4,5]+p_1p_3p_5 = 360+1680+10\cdot 12\cdot 7 = 2880 \\ M[2,2]+M[3,5]+p_1p_2p_5 = 0+1140+10\cdot 3\cdot 7 = 1350 \end{cases} = \mathbf{1350}$$

1	2	3	4	5	
0	120	264			1
	0	360	1320		2
		0	720	1140	3
			0	1680	4
				0	5

$\Rightarrow$

1	2	3	4	5	
0	120	264	1080		1
	0	360	1320	1350	2
		0	720	1140	3
			0	1680	4
				0	5

Now product of 5 matrices

$$M[1,5]=\min\begin{cases} M[1,4]+M[5,5]+p_0p_4p_5 = 1080+0+4\cdot 20\cdot 7 = 1544 \\ M[1,3]+M[4,5]+p_0p_3p_5 = 264+1680+4\cdot 12\cdot 7 = 2016 \\ M[1,2]+M[3,5]+p_0p_2p_5 = 120+1140+4\cdot 3\cdot 7 = 1344 \\ M[1,1]+M[2,5]+p_0p_1p_5 = 0+1350+4\cdot 10\cdot 7 = 1630 \end{cases} = \mathbf{1344}$$

1	2	3	4	5	
0	120	264	1080		1
	0	360	1320	1350	2
		0	720	1140	3
			0	1680	4
				0	5

$\Rightarrow$

1	2	3	4	5	
0	120	264	1080	1344	1
	0	360	1320	1350	2
		0	720	1140	3
			0	1680	4
				0	5

COMPARISON WITH DYNAMIC PROGRAMMING

It usually outperforms a top-down memorized algorithm by constant factor, because there is no over-head for recursion and fewer overheads for maintaining the table. In situations where not every subproblem is computed, memorization only solves those that are needed but dynamic programming solves all the subproblems.

In summary, the matrix-claim multiplication problem can be solved in $O(n^3)$ time by either a top-down, memorized algorithm or a bottom-up dynamic-programming algorithm.

LONGES COMMON SUBSEQUENCE (LCS)

A subsequence of a identified sequence is given sequence with some elements left out. Given two sequences X and Y, we say that a sequence Z is a common sequence of X and Y if Z is a subsequence of both X and Y.

In the longest common subsequence problem, we are given two sequences X = (x_1, x_2… and x_m) and Y = (y_1, y_2… y_n) and wish to find a maximum length common subsequence of X and Y. LCS problem can be solved using dynamic programming.

CHARACTERIZING A LONGEST COMMON SUBSEQUENCE

A **Brute-force approach** to solving the LCS problem is to specify all subsequences of X and check each subsequence found. Each subsequence of X corresponds to a subset of the indices {1, 2, ….m} of X, there are 2^m subsequences of X, so this approach requires exponential time.

The LCS problem has an optimal-substructure property. Given a sequence X = (x_1, x_2, … x_m), we define the ith prefix of X, for i = 0, 1, 2…m as X_i = (x_1, x_2…x_i). For example, if X = (A, B, C, B, C, A, B, C) then X_4 = (A, B, C, B).

THEOREM (OPTIMAL SUBSTRUCTURE OF AN LCS)

Let X = (x_1, x_2… x_m) and Y = (y_1, y_2…y_n) be the sequences and let Z = (z_1, z_2…. Z_k) be any LCS of X and Y.

1. If $x_m = y_n$, then $z_k = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y.
3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}.

The above theorem implies that there are either one or two subproblems to examine when finding an LCS of X = $(x_1, x_2, \ldots x_m)$ and Y = $(y_1, y_2 \ldots. Y_n)$. If $x_m = y_n$ we must find an LCS of X_{m-1} and Y_{n-1}. If $x_m \neq y_n$, then we must solve two subproblems finding as LCS of X_{m-1} and Y and finding an LCS of X and Y_{n-1}. Whenever of these LCS's longer is an LCS of X and Y. but each of these subproblems has the subproblems of finding the LCS of X_{m-1} and Y_{n-1}.

Let us defined c[i, j] to be the length of an LCS of the sequence X_i and Y_j. If either i = 0 or j = 0, one of the sequences has length 0, so the LCS has length 0. The optimal substructure of the LCS problem gives the recurrence formula.

$$c[i,j]=\begin{cases} 0 & \text{if } i=0 \quad \text{or} \quad j=0 \\ c[i-1,j-1]+1 & \text{if } i,j>0 \text{ and } x_i = y_j \\ \max(c[i,j-1],c[i-1,j]) & \text{if } i,j>0 \text{ and } x_i \neq y_j \end{cases}$$

Example: Given two sequences X[1..m] and Y[1..n]. Find the longest subsequences common to both to both. Note: not substring, subsequence.

So if x: A B C B D A B

Y: B D C A B A

The longest subsequence turns out to be B C B A

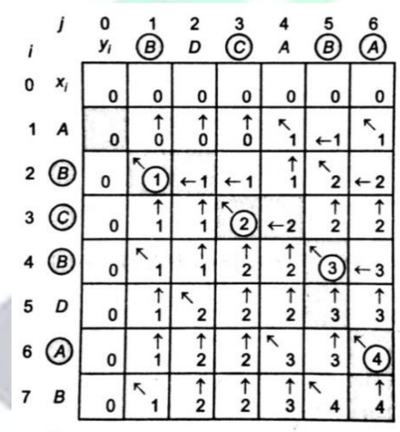
Here X = (A, B, C, B, D, A, B) and Y = (B, D, C, A, B, A)

m = length [X] and n = length[Y]

m = 7 and n = 6

Now, filling in the m × n table with the value of c[i, j] and the appropriate arrow for the value of b[i, j]. Initialize top now and left column to 0 which takes $\theta(m + n)$ time.

Work across the rows starting at the top. Any time $x_i = y_i$ fill in the diagonal neighbor + 1 and mark the box with the wingding __ otherwise fill in the box with the max of the box above and box to the left. That is, the entry of c[i, j] depends only on whether $x_i = y_j$ and the values in entries c[i - 1, j], c[i, j - 1] which are computed before c[i, j]. The max length is the lower right hand corner. In c[i - 1, j] and c[i, j - 1] entries if c[i - 1, j] $\geq$ c[i, j - 1] then b[i, j] entry is '↑' otherwise "←".



BACKTRACKING ALGORITHMS

Backtracking algorithms are based on a depth-first recursive search. A backtracking algorithms:

- Tests to see if a solution has been found, and if so, returns it; otherwise
- For each choice that can be made at this point,
  1. Make that choice
  2. Recur
  3. If the recursion returns a solution, return it
- If no choice remain, return failure.

Example, To color a map with no more than four colurs:

Color (Country n):

If all countries have been colored (n > number of countries) return success ; otherwise,

For each color c four colours,

If country n is not adjacent to country that has been colored c

Color country n with color c

Recursively color country n + 1

If successful, return success

If loop exists, return failure

GREEDY ALGORITHMS

INTRODUCTION

It solve problems by making the choice that seems best at the particular moment. Many optimization problems can be solved using a greedy algorithms. Some problems have no efficient solution, but it provide a solution that is close to optimal. It works if a problem exhibits the following two properties:

1. Greedy choice property. A globally optimal solution can be arrived at by making a locally optimal solution. In other words, an optimal solution can be obtained by making “greedy” choices.
2. Optimal substructure. Optimal solutions contain optimal sub solutions.

AN ACTIVITY-SELECTION PROBLEM

Our first example is the problem of scheduling a resource among several competing activities. We shall find that a greedy algorithm provides a well-designed and simple method for selecting a maximum-size set of mutually compatible activities.

Suppose S = {1, 2, … n} is the set of n proposed activities. The activities share a resource, which can be used by only activity at a time e.g., a Tennis Court, a Lecture Hall etc. Every activity i has a start time s_i and a finish time f_i, where $s_i \le f_i$. If selected, activity i takes place during the half-open time interval $[s_i, f_i]$ do not overlap (i.e., i and j are compatible if $s_i \ge f_j$ or $s_j \ge f_i$). The activity-selection problem selects the maximum-size set of mutual compatible activities.

In this strategy we first select the activity with minimum duration (f_i - s_i) and schedule it. Then, we skip all activities that are no compatible to this one, which means we have to select compatible activities that are not compatible to this one, which means we have to select compatible activity having minimum duration and then we have to schedule it. Thus process is repeated until all the activities are considered. If can be observed that the process of selecting the activity becomes faster it we assume that the input activities are in order by increasing finishing time: $f_1 \le f_2 \le f_3 \le .... \le f_n$.

The running time of algorithm GREEDY-ACTIVITY-SELECTOR is θ(n lg n) as sorting can be done in O(n lg n). there are O(1) operations per activity, thus total time is

$$O(n \lg n) + n.O(1) = O(n \lg n).$$

Example: Given 10 activities along with their start and finish time as

$$S = (A_1, A_2, A_3, A_4, A_5, A_6, A_7, A_8, A_9, A_{10})$$

$$S_i = (1, 2, 3, 4, 7, 8, 9, 9, 11, 12)$$

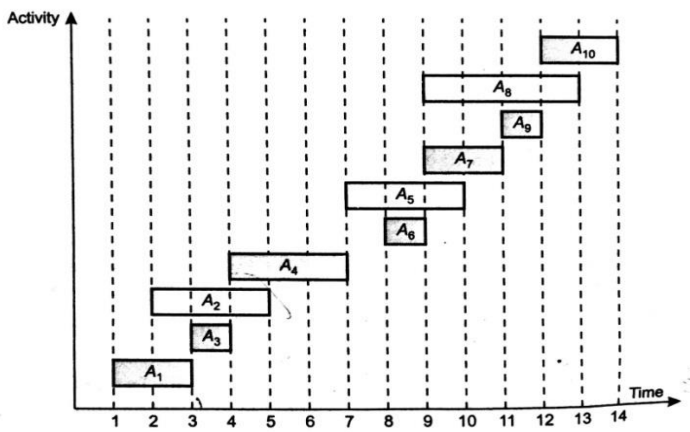
$$f_i = (3, 5, 4, 7, 10, 9, 11, 13, 12, 14)$$

calculate a schedule where the highest number of activities takes place.

Solution: The solution for the above activity scheduling problem using greedy strategy is illustrated below.

order the activities in increasing order of finish time.

Activity	A_1	A_3	A_2	A_4	A_6	A_5	A_7	A_9	A_8	A_{10}
Start	1	3	2	4	8	7	9	11	9	12
Finish	3	4	5	7	9	10	11	12	13	14



Now, schedule A_1

Next, schedule A_3, as A_1 and A_3 are non-interfering

Next, Skip A_2, as it is interfering.

Next, schedule A_4 as A_1, A_3 and A_4 are non-interfering, then next, schedule A_6 as A_1, A_3, A_4 and A_6 are non-interfering.

Skip A_5 as it is interfering.

Next, schedule A_7 as A_1, A_3, A_4, A_6, A_7 are non-interfering.

Next, schedule A_9 as A_1, A_3, A_4, A_6, A_7, and A_9 are non-interfering.

Skip A_8, as it is interfering.

Next, schedule A_{10} as A_1, A_3, A_4, A_6, A_7, A_9 and A_{10}.

KNAPSACK PROBLEMS

We want to pack n items in your luggage.

- The ith item is worth v_i dollars and weight w_i pounds.
- Take as valuable a load as possible, but cannot exceed W pounds.
- v_i, w_i, W are integers.

O-1 KNASACK PROBLEM

- each item is taken or not taken
- Cannot take a fractional amount of an item or take an item more than once.

FRACTIONAL KNAPSACK PROBLEM

- Fractions of items can be taken rather than having to make a binary (0-1) choice for each item.

Both exhibits the optimal-substructure property.

0-1 knapsack problem. Consider a optimal solution. If item j is removed from the load, the remaining load must be the most valuable load weighing at most W - w_j.

Fractional knapsack. If w of item j is removed from the optimal load, the remaining load must be the most valuable load weighing at most W - w that can be taken from other n - 1 items plus w_j - w of item j.

DIFFERENCE BETWEEN GREEDY AND DYNAMIC PROGRAMMING

Because the optimal-substructure property is shown by both greedy and dynamic-programming strategies, one might be tempted to generate a dynamic-programming solution to a problem when a greedy solution suffices, or one might mistakenly think that a greedy solution works when in fact a dynamic-programming solution is required. The most important difference greedy algorithms and dynamic programming is that we don't solve every optimal sub-problem with greedy algorithms. In some cases, Greedy algorithms can be sued to produce sub-optimal solutions. That is solutions which aren't necessarily optimal, but are perhaps very close.

In dynamic programming, we make a choice at step, but the choice may depend on the solutions to sub-problems. In this, we make whatever choice seems best at the moment and then solve the sub-problem, arising after the choice is made. The choice made by a greedy algorithm may depend on choices so far, but it cannot depend on any further choices or an the solutions to sub-problems. Thus, unlike dynamic programming, which solves the sub-problems bottom up, a greedy strategy usually progresses in a top-down fashion, making one greedy choice after another, interactively reducing each given problem instance to a smaller one.

Fractional knapsake problem can be solvable by the greedy strategy whereas the 0-1 problem is not. To solve the fractional problem.

- Compute the value per pound v_i / w_i for each item
- Obeying a greedy strategy, we take as much as possible of the item with the greatest value per pound.
- If the supply of that item is exhausted and we can still carry more, we take as much as possible of the item with the next value per pound, and so forth until we cannot carry any more.
- Sorting the items by value per pound, the greedy algorithm runs in O(n lg n) time.

0-1 knapsack problem cannot be solved by the greedy strategy because it is unable to fill the knapsake capacity, and the empty space lowers the effective value per pound of the load and we must compare the solution to the sub-problem in which the item is included with the solution to the sub-problem in which the item is excluded before we can make the choice.

Example: Consider 5 items along their respective weights and values.

$$I = (I_1, I_2, I_3, I_4, I_5)$$

$$w = (5, 10, 20, 30, 40)$$

$v = (30, 20, 100, 90, 160)$

The capacity of knapsack W = 60. Find the solution to the fractional knapsack problem.

Solution: Initially,

Item	w_i	v_i
I_1	5	30
I_2	10	20
I_3	20	100
I_4	30	90
I_5	40	160

Taking value per weight ratio i.e., $p_i = v_i / w_i$

Item	w_i	v_i	$p_i = v_i / w_i$
I_1	5	30	6.0
I_2	10	20	2.0
I_3	20	100	5.0
I_4	30	90	3.0
I_5	40	160	4.0

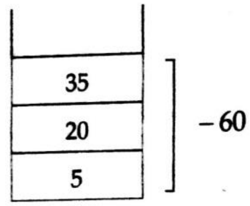
Now arrange the value of p_i in decreasing order

Item	w_i	v_i	$p_i = v_i / w_i$
I_1	5	30	6.0
I_3	20	100	5.0
I_5	40	160	4.0
I_4	30	90	3.0
I_2	10	20	2.0

Now, fill the knapsack according to the decreasing value of p_i.

First we choose item I_1 whose weight is 5, then choose item I_3 whose weight is 20. Now the total weight in knapsack is 5 + 20 = 25.

Now, the next item is I_5 and its weight is 40, but we want only 35. So we choose fractional part of if i.e.,



The value of fractional part of I_5 is

$$\frac{160}{40} \times 35 = 140$$

Thus the maximum value is

$$= 30 + 100 + 120 = 270.$$