Dynamic Programming

By

Prasanta K. Jana, ***IEEE Senior Member***

Department of Computer Science and Engineering
Indian Institute of Technology (ISM), Dhanbad
E-mail: prasantajana@iitism.ac.in

Dynamic Programming

1) Solution: Result of a sequence of decisions.

2) Why Dynamic Programming,

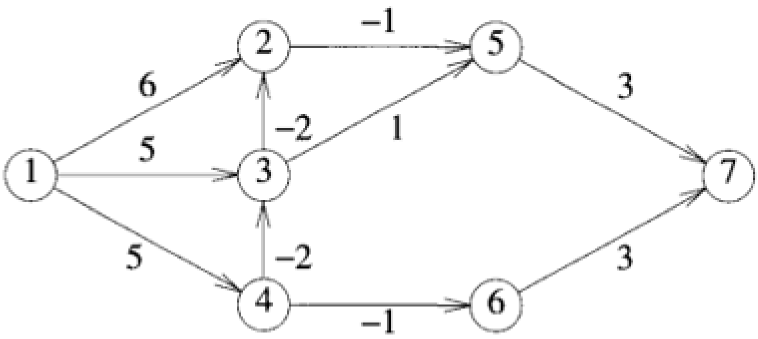
3) Principle of optimality,

4) Difference between DP & Greedy Method.

5) Similar to divide and conquer, DP solves a problem by combining solutions of subproblems

- Subproblems are independent for D&C so more works as it solves common subproblems
- In contrast the subproblems are not independent rather they are overlapped in DP.
  So, DP solves every subproblem just once and saves its answer in a table, thereby avoiding re-computation of the answer every time the subproblem is encountered.

Single Source Shortest Paths



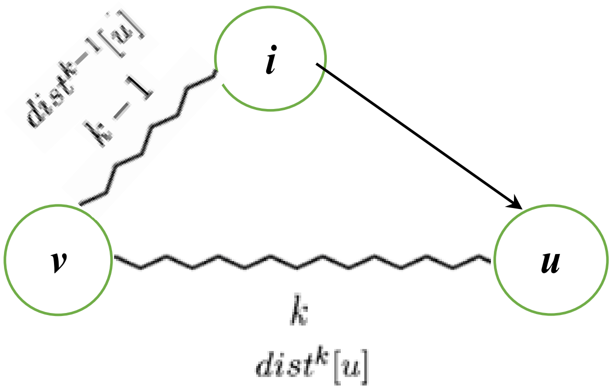
(a) A directed graph

Note: If there is no cycle of negative length, any shortest path has at most $n - 1$ edges for a graph of nodes n.

$\text{dist}^k[u]$: Length of the shortest path from source v to u having at most k edges.

$\therefore \text{dist}^1[u] = \text{cost}(v, u)$

We have to find out $\text{dist}^{n-1}[u]\ \forall\ u$.



1. If the shortest path from v to u with at most k, $k > 1$, edges has no more than $k-1$ edges, then $dist^k[u] = dist^{k-1}[u]$.

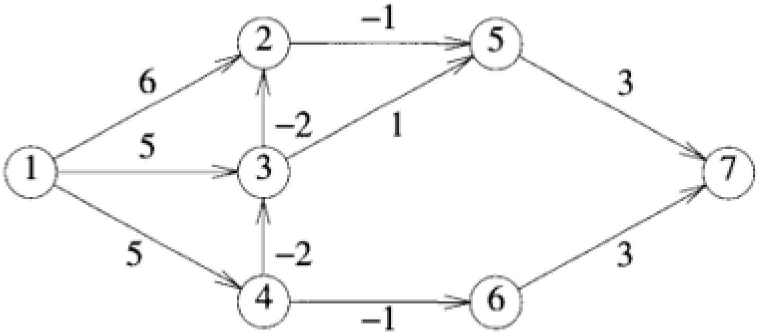
2. If it has exactly *k* edges,

then $dist^k[u] = dist^{k-1}[i] + cost[i, u]$

$$dist^k[u] = \min \{dist^{k-1}[u], \min_i \{dist^{k-1}[i] + cost[i, u]\}\}$$

This recurrence can be used to compute $dist^k$ from $dist^{k-1}$, for $k = 2, 3, \ldots,$ $n-1$.

An Illustration:



(a) A directed graph

k	$dist^k[1..7]$						
	1	2	3	4	5	6	7
1	0	6	5	5	∞	∞	∞
2	0	3	3	5	5	4	∞
3	0	1	3	5	2	4	7
4	0	1	3	5	0	4	5
5	0	1	3	5	0	4	3
6	0	1	3	5	0	4	3

(b) $dist^k$

```
Algorithm BellmanFord(v, cost, dist, n)
// Single-source/all-destinations shortest
// paths with negative edge costs
{
    for i := 1 to n do // Initialize dist.
        dist[i] := cost[v, i];
    for k := 2 to n − 1 do
        for each u such that u ≠ v and u has
                at least one incoming edge do
A           for each ⟨i, u⟩ in the graph do
                if dist[u] > dist[i] + cost[i, u] then
                    dist[u] := dist[i] + cost[i, u];
}
```

A $\longrightarrow$ requires $O(n^2)$ for adjacency matrix (or) $O(e)$ for adjacency list.

$\therefore$ Time complexity: $O(n^3)$ for adjacency matrix, $O(ne)$ for adjacency list.

All Pairs Shortest Path

(Floyd Warshall Algorithm)

Statement: To determine a matrix A such that A(*i, j*) is the length of the shortest path from *i* to *j*.

Assumption: No negative cycles.

Let $A^k(i,j)$ be the shortest path going through no vertex higher than k.

Then, $A^0(i, j) = \text{cost}\ (i, j)$

If the shortest path goes through *k*

$$A^k(i,j) = A^{k-1}(i,k) + A^{k-1}(k,j) \longrightarrow \quad (1)$$

If not

$$A^k(i,j) = A^{k-1}(i,j) \longrightarrow \quad (2)$$

Combining (1) and (2), we get

$$A^k(i,j) = \min\ \{A^{k-1}(i,j),\ A^{k-1}(i,k) + A^{k-1}(k,j)\}$$

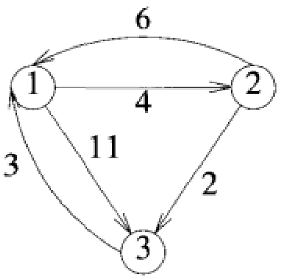
```
Algorithm AllPaths(cost, A, n)
// cost[1 : n, 1 : n] is the cost adjacency matrix of a graph with
// n vertices; A[i, j] is the cost of a shortest path from vertex
// i to vertex j. cost[i, i] = 0.0, for 1 ≤ i ≤ n.
{
    for i := 1 to n do
        for j := 1 to n do
            A[i, j] := cost[i, j]; // Copy cost into A.
    for k := 1 to n do
        for i := 1 to n do
            for j := 1 to n do
                A[i, j] := min(A[i, j], A[i, k] + A[k, j]);
}
```

Time Complexity: $O(n^3)$

An Illustration:

Let A =

$$\begin{matrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \infty & 0 \end{matrix}$$



A^0	1	2	3
1	0	4	11
2	6	0	2
3	3	∞	0

(b) A^0

A^1	1	2	3
1	0	4	11
2	6	0	2
3	3	7	0

(c) A^1

A^2	1	2	3
1	0	4	6
2	6	0	2
3	3	7	0

(d) A^2

A^3	1	2	3
1	0	4	6
2	5	0	2
3	3	7	0

(e) A^3

Longest Common Subsequence Problem

Subsequence: Z = (B, C, D, B) is a subsequence of X =(A, B, A, C, D,A,D,A,B)

Common subsequence: (B, C, A) is a common subsequence of

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

Longest common subsequence: The common subsequence of two sequences with largest length.

(B, C, A, B) is a LCS of X and Y

(B, C, B, D) is another LCS of X and Y.

NOTE – Thus LCS of two sequence is not unique.

Longest-common-subsequence problem: Given two sequences X= (x_1, x_2,…, x_m) and Y = (y_1, y_2,….., y_n) and wish to find a maximum length common subsequence of X and Y.

Brute Force Approach:

Step1: Enumerate all subsequence of X.

Step2: check each subsequence to see if it is also subsequence of Y.

Step3: Find the largest one of them in step 2.

Time Complexity:

1) $\exists\ 2^m$ subsequences of X

2) So, it has exponential time complexity

Dynamic Programming Approach:

NOTE –LCS problem has optimal substructure property as noted by the following theorem

$X = (x_1, x_2, \ldots, x_m)$ and $Y = (y_1, y_2, \ldots, y_n)$

$X = (x_1, x_2, \ldots, x_i)$, $i = 0, 1, \ldots, m$ is defined as the i^{th} prefix of X.

e.g. X = (A, B, C, B, D) then X_4 = (A, B, C, B), X_0 is the empty sequence.

$Z = (z_1, z_2, \ldots, z_k)$ be any LCS X and Y.

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

A recursive solution to problem:

Theorem implies that there are either one or two subproblems when finding an LCS of X and Y.

1)If $x_m = y_n$ then we are to find an LCS of X_{m-1} and $Y_{n-1.}$

2)If $x_m \neq y_n$ then we are to solve two subproblems

2.1) we have to find LCS of X_{m-1} and Y.

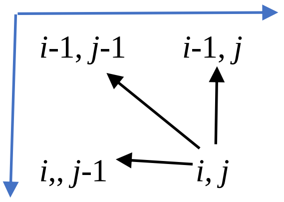
2.2) we have to find LCS of X and $Y_{n-1.}$

Whichever is longer will be LCS of X and Y.

Therefore, if c[i, j] denotes the length of LCS of X_i and Y_j. then:

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

NOTE – if either i=0 or j=0, one of the sequence has length 0, so the LCS has length 0.



Computing the length of the LCS

```
LCS-LENGTH(X, Y)
m = X.length
n = Y.length
let b[1..m, 1..n] and c[0..m, 0..n] be new tables
for i = 1 to m
    c[i, 0] = 0
for j = 0 to n
    c[0, j] = 0
for i = 1 to m
    for j = 1 to n
        if x_i == y_j
            c[i, j] = c[i-1, j-1] + 1
            b[i, j] = "↖"
        elseif c[i-1, j] ≥ c[i, j-1]
            c[i, j] = c[i-1, j]
            b[i, j] = "↑"
        else c[i, j] = c[i, j-1]
            b[i, j] = "←"
return c and b
```

Time complexity: entries are captured in row-major order from left to right, and each entry take O(1) time to execute. Since there are ‘m’ rows and ‘n’ columns it require O(m*n) time to execute.

An Illustration:

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

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑0	↑0	↑0	↖1	←1	↖1
2	B	0	↖1	←1	←1	↑1	↖2	←2
3	C	0	↑1	↑1	↖2	←2	↑2	↑2
4	B	0	↖1	↑1	↑2	↑2	↖3	←3
5	D	0	↑1	↖2	↑2	↑2	↑3	↑3
6	A	0	↑1	↑2	↑2	↖3	↑3	↖4
7	B	0	↖1	↑2	↑2	↑3	↖4	↑4

We simply begin at b[m,n] and trace through the table by following the arrows. Whenever we encounter a "↖" in entry b[i,j] it implies that $x_m = y_n$ is an element of the LCS .

NOTE : the element of the LCS are encountered in reverse order using this method.

Thus following the "↖" we got BCBA as the LCS.

Following procedure runs it

```
PRINT-LCS(b, X, i, j)
if i == 0 or j == 0
    return
if b[i, j] == "↖"
    PRINT-LCS(b, X, i − 1, j − 1)
    print x_i
elseif b[i, j] == "↑"
    PRINT-LCS(b, X, i − 1, j)
else PRINT-LCS(b, X, i, j − 1)
```

Time Complexity: Since at least one of i and j is decremented at each stage in the recursion, it requires O(m+n).

Overall Time Complexity: O(m*n)

Overall Space Complexity: O(m*n)

Tutorial: Determine an LCS of <1,0,0,1,0,1,0,1> and <0,1,0,1,1,0,1,1,0>
Ans : 100110

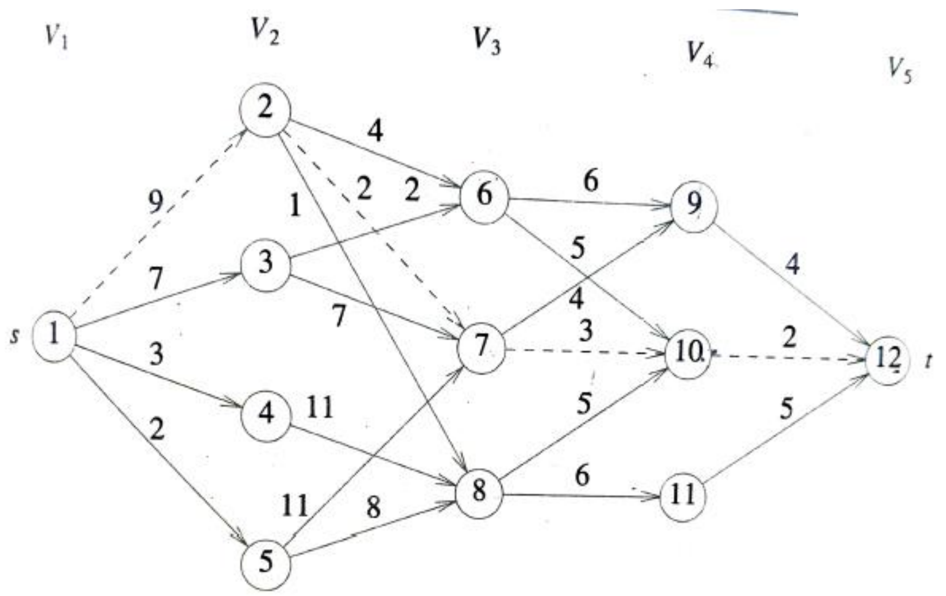
Multistage graph problem

A multistage graph: A directed graph in which vertices are partitioned into k >= 2 disjoint sets $V_1, V_2, \ldots, V_k$ such that

1) $|V_1| = |V_k| = 1$

2) If there exists an edge <u, v> then if $u \in V_i$ then v must belong to V_{i+1}.

V_1: source (s) and V_k: sink (t)

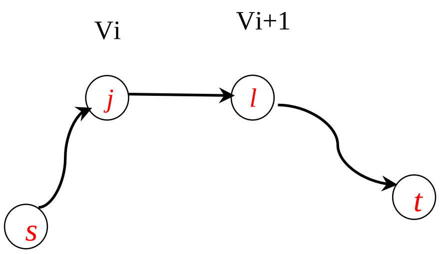


Problem Statement: To determine a minimum-cost path from s to t.

Let c[i, j] be the cost of the edge <i, j>

Let P(i, j) be the minimum cost path from vertex j in V_i to t and

Let cost (i, j) be its cost.

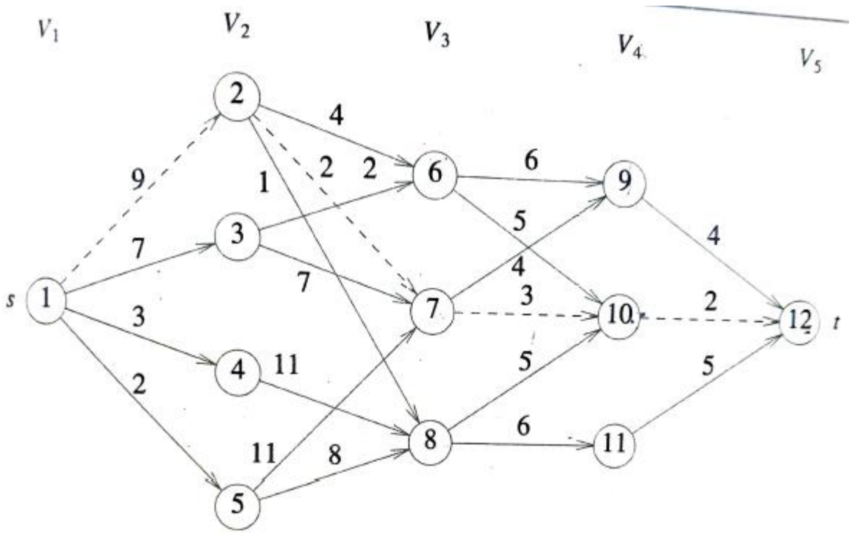


Initialization: cost (k-1, j) = c[j, t]

Solution: We have to find the Cost(1, s).

Intermediate Calculations:

$$cost(i, j) = \min_{\substack{l \in V_{i+1} \\ \langle j, l\rangle \in E}} \{c(j, l) + cost(i+1, l)\}$$



$$cost(i, j) = \min_{\substack{l \in V_{i+1} \\ (j,l) \in E}} \{c(j, l) + cost(i + 1, l)\}$$

Let d(*i*, *j*) be the value of *l* that minimizes Cost(*i*+1, *l*) + c(*j*, *l*).

Then d(3, 6) = 10, d(3, 7) = 10, d(3, 8) = 10;

d(2, 2) = 7, d(2, 3) = 6, d(2, 4) = 8, d(2, 5) = 8;

d(1, 1) = 2

Then minimum cost path is 1, v1, v2, …, vk-1, t.

Here, v2 = d(1, 1) = 2;

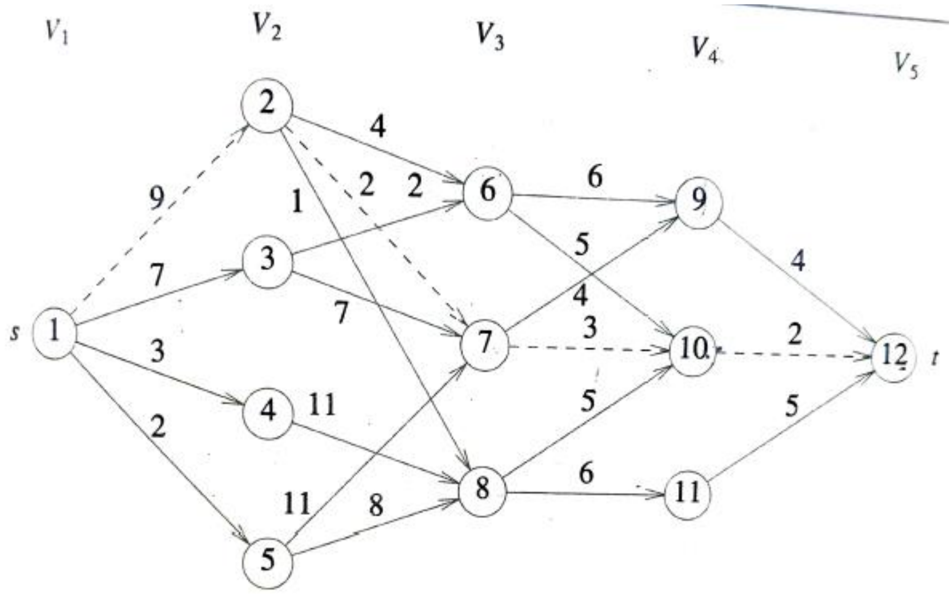
v3 = d(2, d(1, 1)) = d(2, 2) = 7

v4 = d(3, d(2, d(1, 1))) = d(3, 7) = 10

Therefore, the minimum cost path is 1, 2, 7, 10, 12

Algorithm using Forward Approach:

$$cost(i,j) = \min_{\substack{l \in V_{i+1} \\ \langle j,l \rangle \in E}} \{c(j,l) + cost(i+1,l)\}$$



Algorithm FGraph(G, k, n, p)
// The input is a k-stage graph $G = (V, E)$ with n vertices
// indexed in order of stages. E is a set of edges and $c[i,j]$
// is the cost of $\langle i,j \rangle$. $p[1:k]$ is a minimum-cost path.
{
    $cost[n] := 0.0$;
    for $j := n-1$ **to** 1 **step** -1 **do**
    { // Compute $cost[j]$.
        Let r be a vertex such that $\langle j,r \rangle$ is an edge
        of G and $c[j,r] + cost[r]$ is minimum;
        $cost[j] := c[j,r] + cost[r]$;
        $d[j] := r$;
    }
    // Find a minimum-cost path.
    $p[1] := 1$; $p[k] := n$;
    for $j := 2$ **to** $k-1$ **do** $p[j] := d[p[j-1]]$;
}

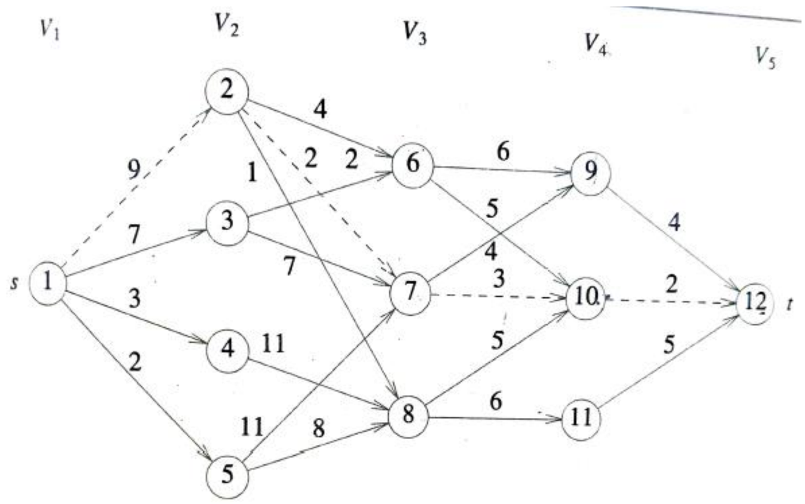
Time Complexity: If the graph is represented by **adjacency list**, then it is:

$$\Theta(|V| + |E|)$$

Solution Using Backward Approach:

The multistage graph problem can also be solved using the backward approach. Let $bp(i, j)$ be a minimum-cost path from vertex s to a vertex j in V_i. Let $bcost(i, j)$ be the cost of $bp(i, j)$. From the backward approach we obtain

$$bcost(i, j) = \min_{\substack{l \in V_{i-1} \\ \langle l, j \rangle \in E}} \{bcost(i-1, l) + c(l, j)\}$$



$$\begin{aligned}
bcost(3, 6) &= \min\ \{bcost(2, 2) + c(2, 6), bcost(2, 3) + c(3, 6)\} \\
&= \min\ \{9 + 4, 7 + 2\} \\
&= 9 \\
bcost(3, 7) &= 11 \\
bcost(3, 8) &= 10 \\
bcost(4, 9) &= 15
\end{aligned}$$

$$\begin{aligned}
bcost(4, 10) &= 14 \\
bcost(4, 11) &= 16 \\
bcost(5, 12) &= 16
\end{aligned}$$

```
Algorithm BGraph(G, k, n, p)
// Same function as FGraph
{
    bcost[1] := 0.0;
    for j := 2 to n do
    { // Compute bcost[j].
        Let r be such that ⟨r, j⟩ is an edge of
        G and bcost[r] + c[r, j] is minimum;
        bcost[j] := bcost[r] + c[r, j];
        d[j] := r;
    }
    // Find a minimum-cost path.
    p[1] := 1; p[k] := n;
    for j := k - 1 to 2 do p[j] := d[p[j + 1]];
}
```

Travelling Sales Person Problem

Let $g(i, S)$ be the length of a shortest path starting at vertex i, going through all vertices in S, and terminating at vertex 1.

Then $g(1, V - \{1\})$ is the length of an optimal salesperson tour.

From the principle of optimality,

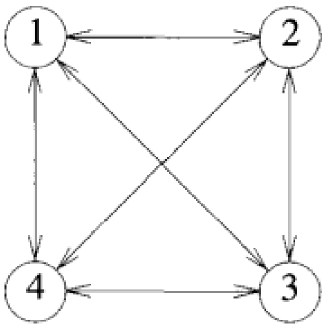
$$g(1, V - \{1\}) = \min_{2 \le k \le n} \{c_{1k} + g(k, V - \{1, k\})\} \longrightarrow (1)$$

Generalizing (1) , we obtain (for $i \notin S$)

$$g(i, S) = \min_{j \in S}\{c_{ij} + g(j, S - \{j\})\} \longrightarrow (2)$$

Equation (1) can be solved for $g(1, V - \{1\})$ if we know $g(k, V - \{1, k\})$ for all choices of k. The g values can be obtained by using (2)). Clearly,

$$g(i, \phi) = c_{i1},\ 1 \le i \le n.$$



(a)

$$\begin{bmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{bmatrix}$$

(b)

Thus $g(2, \phi) = c_{21} = 5, g(3, \phi) = c_{31} = 6$, and $g(4, \phi) = c_{41} = 8$. Using ((2)), we obtain

$$\begin{aligned} g(2, \{3\}) &= c_{23} + g(3, \phi) = 15 & g(2, \{4\}) &= 18 \\ g(3, \{2\}) &= 18 & g(3, \{4\}) &= 20 \\ g(4, \{2\}) &= 13 & g(4, \{3\}) &= 15 \end{aligned}$$

Next, we compute $g(i, S)$ with $|S| = 2$, $i \neq 1$, $1 \notin S$ and $i \notin S$.

$$\begin{array}{lclcl} g(2, \{3, 4\}) & = & \min \ \{c_{23} + g(3, \{4\}), c_{24} + g(4, \{3\})\} & = & 25 \\ g(3, \{2, 4\}) & = & \min \ \{c_{32} + g(2, \{4\}), c_{34} + g(4, \{2\})\} & = & 25 \\ g(4, \{2, 3\}) & = & \min \ \{c_{42} + g(2, \{3\}), c_{43} + g(3, \{2\})\} & = & 23 \end{array}$$

Finally, from '(1)' we obtain

$$\begin{array}{lcl} g(1, \{2, 3, 4\}) & = & \min \{c_{12} + g(2, \{3, 4\}), c_{13} + g(3, \{2, 4\}), c_{14} + g(4, \{2, 3\})\} \\ & = & \min \{35, 40, 43\} \\ & = & 35 \end{array}$$

An optimal tour of the graph of has length 35.

A tour of this length can be constructed if we retain with each g(i,S) the value of j that minimizes the right-hand side of (2).

Let J(i,S) be this value.

Then, $J(1, \{2, 3, 4\}) = 2$.

$J(2, \{3, 4\}) = 4$.

$J(4, \{3\}) = 2$.

The optimal tour is 1, 2, 4, 3, 1.

Time Complexity:

Let N be the number of $g(i, S)$'s that have to be computed before (1) can be used to compute $g(1, V - \{1\})$. For each value of $|S|$ there are choices for i. The number of distinct sets S of size k not including 1 and i is $\binom{n-2}{k}$. Hence

$$N = \sum_{k=0}^{n-2} (n-1)\binom{n-2}{k} = (n-1)2^{n-2}$$

∴ time complexity is $O(n\,2^{n-2})\, n = O(n^2 * 2^{n-2})$

Note: This is better than $n!$ for all possible tours.
Serious drawback of this D.P solution is the space needed, $O(n2^n)$

MATRIX CHAIN MULTIPLICATION

Consider < A_1 A_2 A_3 > chain where $[A_1]_{10 \times 100}$, $[A_2]_{100 \times 5}$, $[A_3]_{5 \times 50}$.

- $((A_1 A_2) A_3)$ requires $10 \times 100 \times 5 + 10 \times 5 \times 50 = 7500$ scalar multiplications.

- $(A_1(A_2 A_3))$ requires $100 \times 5 \times 50 + 10 \times 100 \times 50 = 75000$ scalar multiplications.

So $((A_1 A_2) A_3)$ is 10 times faster than $(A_1(A_2 A_3))$.

Now consider: < A_1 A_2 A_3 A_4 >

Then several ways to fully parenthesize:

$$(A_1(A_2(A_3A_4)))$$
$$(A_1((A_2A_3)A_4))$$
$$((A_1A_2)(A_3A_4))$$
$$((A_1(A_2A_3))A_4)$$
$$(((A_1A_2)A_3)A_4)$$

Problem Statement: Given a chain < $A_1, A_2, A_3, \ldots, A_n$ > of n matrices, where A_i has dimension $p_{i-1} \times p_i$, for $i = 1\ldots, n$, fully parenthesize the product in a way that minimizes the number of scalar multiplications.

Brute force approach:

P(n) → Number of alternative parenthesizations. Then,

$$P(n) = \begin{cases} 1 & , if\ n\ =\ 1 \\ \sum_{k=0}^{n} P(k)P(n-k) & , if\ n\ >= 2 \end{cases}$$

P(n) = C(n-1) is the solution of this recurrence equation where,

$$C(n) = \frac{1}{n+1}\binom{2n}{n}$$

And C(n) = Ω (4^n / $n^{3/2}$)

Let $A_{i...j}$ denote A_i A_{i+1} …A_j

Let $m[i, j]$ be the minimum number of scalar multiplications needed to compute the product $A_{i...j}$.

Then $m[i, j]$ would be the minimum number of scalar multiplications needed to compute the product $A_{i...k}$ and be the minimum number of scalar multiplications needed to compute the product $A_{k+1...j}$ plus number of scalar multiplications for multiplying them. Therefore,

$\boxed{A_i \text{ has dimension } p_{i-1} \times p_i.}$

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

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

```
MATRIX-CHAIN-ORDER(p)
n ← length[p] − 1
for i ← 1 to n
     do m[i, i] ← 0
for l ← 2 to n                      ▷ l is the chain length.
     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
```

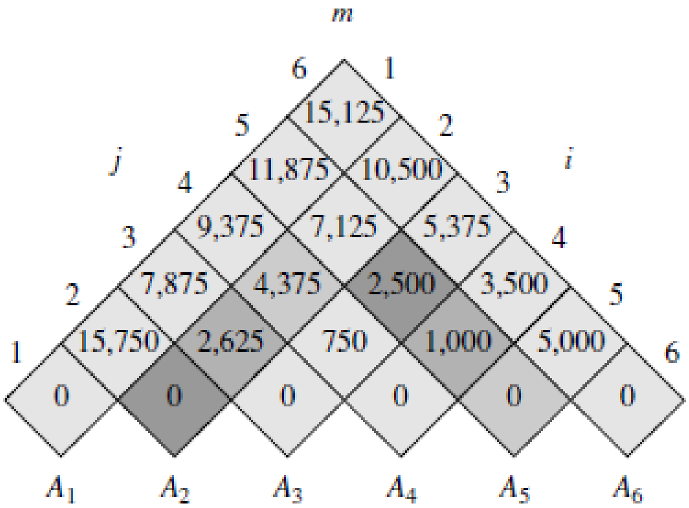
Time Complexity: $O(n^3)$

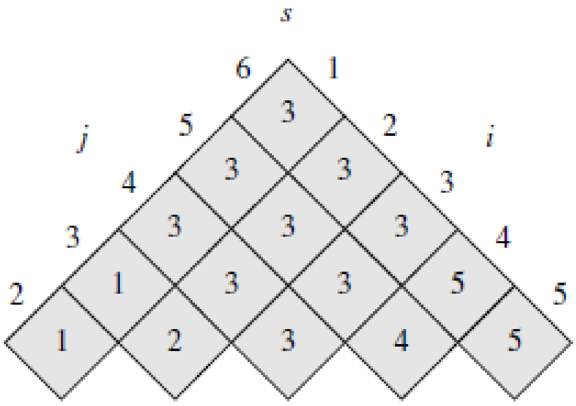
An Illustration:

matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35	35×15	15×5	5×10	10×20	20×25

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

$$m[2,5] = \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 & = 13000 \text{ ,} \\ m[2,3] + m[4,5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 & = 7125 \text{ ,} \\ m[2,4] + m[5,5] + p_1 p_4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 & = 11375 \end{cases}$$
$$= 7125 \text{ .}$$

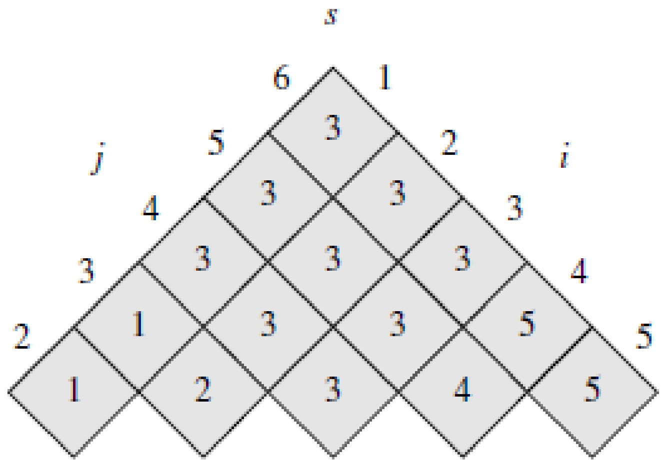


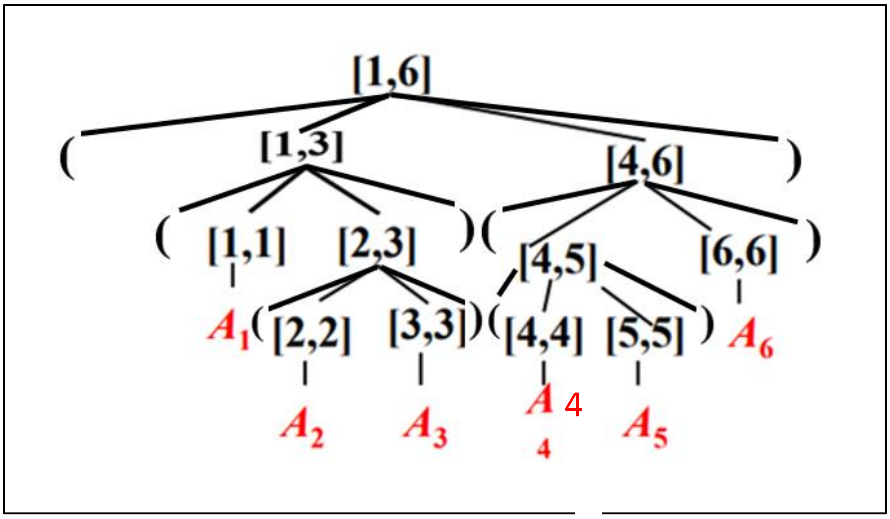


$\ldots, A_2, \ldots, A_n\rangle$.

```
PRINT-OPTIMAL-PARENS(s, i, j)
if i == j
    print "A"_i
else print "("
    PRINT-OPTIMAL-PARENS(s, i, s[i, j])
    PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
    print ")"
```

To be initially called with Print Optimal-Parents(S, 1, n)

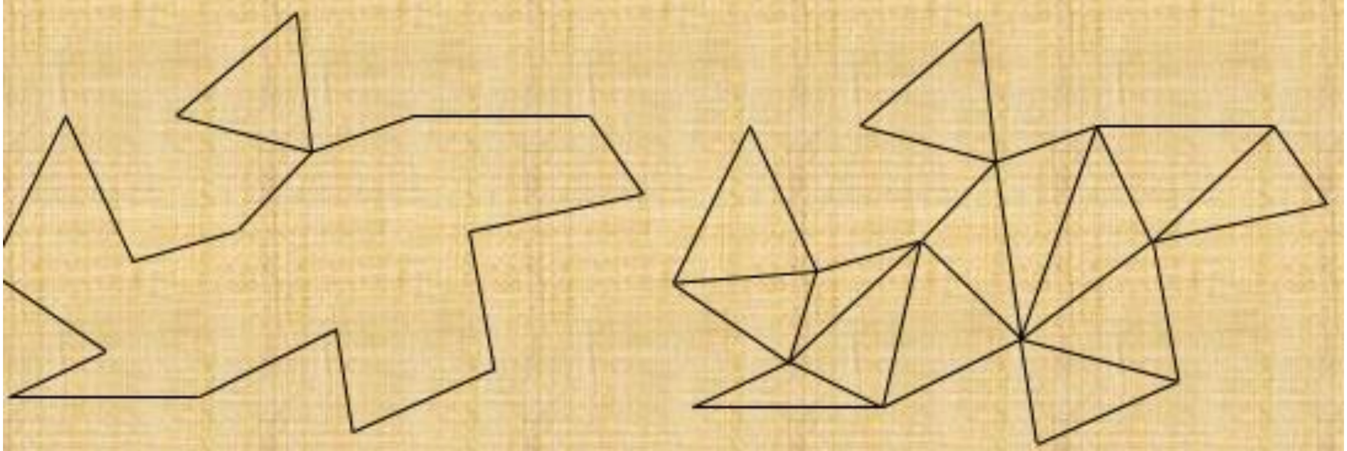




Hence the solution is $((A_1(A_2A_3))((A_4A_5)A_6))$

Optimal Triangulation:

Triangulation: A decomposition of a polygon into triangles by a maximal set of non-intersecting diagonals.



Note: Triangulations are usually not unique.

Theorem: Every simple polygon admits triangulations it consists of exactly n-2 triangles.

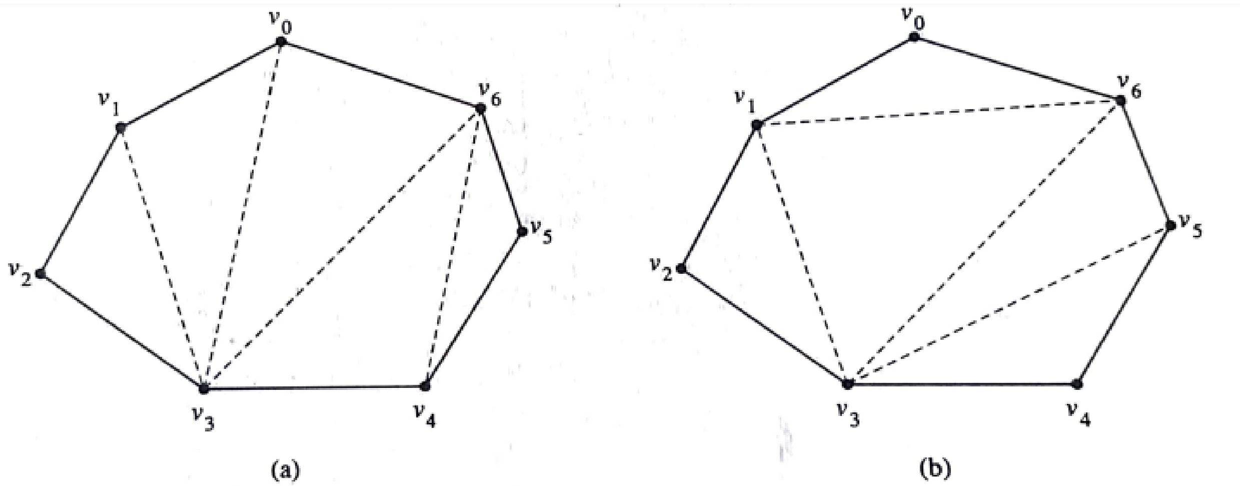


Figure 16.4 Two ways of triangulating a convex`polygon. Every triangulation of this 7-sided polygon has $7 - 3 = 4$ chords and divides the polygon into $7 - 2 = 5$ triangles.

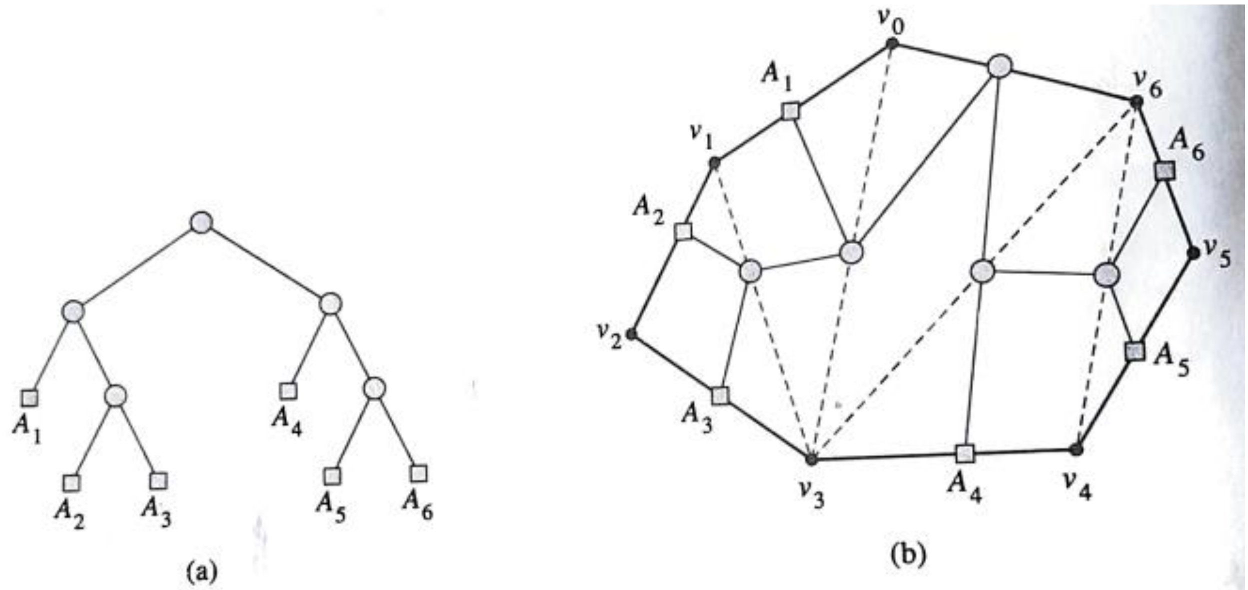


Figure 16.5 Parse trees. **(a)** The parse tree for the parenthesized product $((A_1(A_2A_3))(A_4(A_5A_6)))$ and for the triangulation of the 7-sided polygon from Figure 16.4(a). **(b)** The triangulation of the polygon with the parse tree overlaid. Each matrix A_i corresponds to the side $\overline{v_{i-1}v_i}$ for $i = 1, 2, \ldots, 6$.

$((A_1(A_2A_3))((A_4A_5)A_6))$