Unit-IV

UNIT IV: Dynamic Programming: General method, applications-Matrix chain multiplication, Optimal binary search trees, 0/1 knapsack problem, All pairs shortest path problem, Travelling sales person problem, Reliability design.

Applications: Routing Algorithms in the computer networking

1.Matrix Chain Multiplication

- Matrix-chain multiplication problem
  - Given a chain $A_1, A_2, \ldots, A_n$ of n matrices, where for i=1, 2, …, n, matrix A_i has dimension $p_{i-1} \times p_i$
  - Parenthesize the product $A_1 A_2 \ldots A_n$ such that the total number of scalar multiplications is minimized.

Matrix Multiplication

$$\begin{bmatrix} \boxed{2} & \boxed{5} & \boxed{1} \\ 4 & 2 & 3 \end{bmatrix} \bullet \begin{bmatrix} 2 & 5 \\ 1 & 1 \\ 3 & 2 \end{bmatrix}$$

$p \times q$ $\quad$ $q \times r$

$$= \begin{bmatrix} \boxed{2} \bullet 2 + \boxed{5} \bullet 1 + \boxed{1} \bullet 3 & 4 \bullet 2 + 2 \bullet 1 + 3 \bullet 3 \\ 2 \bullet 5 + 5 \bullet 1 + 1 \bullet 2 & 4 \bullet 5 + 2 \bullet 1 + 3 \bullet 2 \end{bmatrix}_{p \times r}$$

Cost: Number of scalar multiplications = **pqr**

Example

Matrix	Dimensions
A_1	13 x 5
A_2	5 X 89
A_3	89 X 3
A_4	3 X 34

Parenthesization	Scalar multiplications
1 $((A_1 A_2) A_3) A_4$	10,582
2 $(A_1 A_2)(A_3 A_4)$	54,201
3 $(A_1 (A_2 A_3)) A_4$	2, 856
4 $A_1 ((A_2 A_3) A_4)$	4, 055
5 $A_1 (A_2 (A_3 A_4))$	26,418

1. 13 x 5 x 89 *scalar multiplications* to get (A_1A_2) 13 x 89 result
   13 x 89 x 3 *scalar multiplications* to get $((A_1A_2)A_3)$ 13 x 3 result
   13 x 3 x 34 *scalar multiplications* to get $(((A_1A_2)A_3) A_4)$ 13 x 34

Dynamic Programming Approach

- The structure of an optimal solution
  - Let us use the notation $A_{i..j}$ for the matrix that results from the product $A_i A_{i+1} \ldots A_j$
  - An optimal parenthesization of the product $A_1 A_2 \ldots A_n$ splits the product between A_k and A_{k+1} for some integer k where $1 \leq k < n$
  - First compute matrices $A_{1..k}$ and $A_{k+1..n}$; then multiply them to get the final matrix $A_{1..n}$

Dynamic Programming Approach

…contd

- **Key observation**: parenthesizations of the subchains $A_1 A_2 \ldots A_k$ and $A_{k+1} A_{k+2} \ldots A_n$ must also be optimal if the parenthesization of the chain $A_1 A_2 \ldots A_n$ is optimal.

- That is, the optimal solution to the problem contains within it the optimal solution to subproblems.

Dynamic Programming Approach

…contd

- Recursive definition of the value of an optimal solution.
  - Let *m[i, j]* be the minimum number of scalar multiplications necessary to compute $A_{i..j}$
  - Minimum cost to compute $A_{1..n}$ is m[1, n]
  - Suppose the optimal parenthesization of $A_{i..j}$ splits the product between A_k and A_{k+1} for some integer k where $i \leq k < j$

Dynamic Programming Approach

...contd

- $A_{i..j} = (A_i A_{i+1} \ldots A_k) \cdot (A_{k+1} A_{k+2} \ldots A_j) = A_{i..k} \cdot A_{k+1..j}$

- Cost of computing $A_{i..j}$ = *cost of computing* $A_{i..k}$ + *cost of computing* $A_{k+1..j}$ + *cost of multiplying* $A_{i..k}$ *and* $A_{k+1..j}$

- Cost of multiplying $A_{i..k}$ and $A_{k+1..j}$ is $p_{i-1} p_k p_j$

- $m[i, j] = m[i, k] + m[k+1, j] + p_{i-1} p_k p_j \qquad \text{for } i \le k < j$
- $m[i, i] = 0$ for $i = 1, 2, \ldots, n$

Dynamic Programming Approach

…contd

- But… *optimal* parenthesization occurs at one value of k among all possible $i \leq k < j$
- *Check* all these and select the *best* one

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

Dynamic Programming Approach

…contd

- To keep track of how to construct an optimal solution, we use a *table s*
- *s[i, j]* = value of *k* at which $A_i A_{i+1} \ldots A_j$ is split for *optimal* parenthesization.

Ex:-

$[A_1]_{5\times4}$ $[A_2]_{4\times6}$ $[A_3]_{6\times2}$ $[A_4]_{2\times7}$

$P_0=5$, $p_1=4$, $p_2=6$, $p_3=2$, $p_4=7$

Sequence Size		1	2	3	4
	1	$M_{11}=0$	$M_{22}=0$	$M_{33}=0$	$M_{44}=0$
	2	$M_{12}=$ $K=$	$M_{23}=$ $K=$	$M_{34}=$ $K=$	
	3	$M_{13}=$ $K=$	$M_{24}=$ $K=$		
	4	$M_{14}=$ $K=$			

Matrix Chain Multiplication Algorithm

- First computes costs for chains of length l=1
- Then for chains of length l=2,3, … and so on
- Computes the optimal cost bottom-up.

Input: Array $p[0\ldots n]$ containing matrix dimensions and n
Result: Minimum-cost table m and split table s

```
Algorithm  Matrix_Chain_Mul(p[], n)
{
        for i:= 1 to n  do
                m[i, i]:= 0 ;

        for len:= 2 to n  do        // for lengths  2,3 and so on
    {
                for i:= 1 to ( n-len+1 ) do
                {
                        j:= i+len-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]
                                        {
                                                m[i, j]:= q;
                                                s[i, j]:= k;
                                        }
                                }
                }
        }
return m and s
}
```

Time complexity of above algorithm is $O(n^3)$

Constructing Optimal Solution

- Our algorithm computes the minimum-cost table *m* and the split table *s*
- The *optimal solution* can be constructed from the split table *s*
  - Each entry $s[i, j]=k$ shows where to split the product $A_i A_{i+1} \dots A_j$ for the minimum cost.

Example

- Copy the table of previous example and then construct optimal parenthesization.

Optimal Binary Search Tree(OBST)

- *A binary search tree T is a binary tree, either it is empty or each node in the tree contains an identifier and,*
  - All identifiers in the left subtree of T are *less than* the identifier in the *root* node T.
  - All identifiers in the right subtree are *greater than* the identifier in the *root* node T.
  - The *left and right* subtree of T are also *binary search* trees.

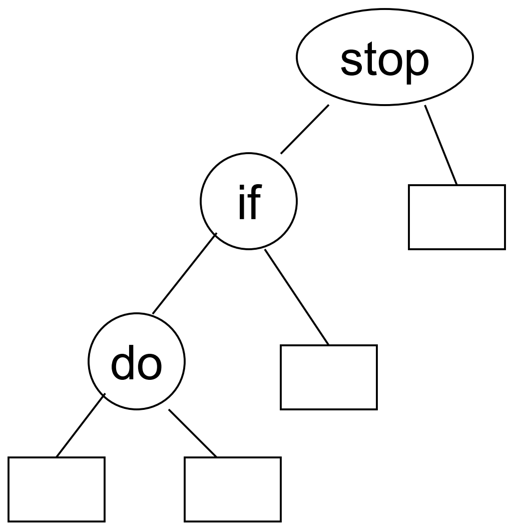
- Ex:- (a1,a2,a3)=(do,if,stop)

Here n=3

- The number of possible binary search trees= $(1/n+1)2n_{cn}$

$= ¼(6c_3)$

$=5$

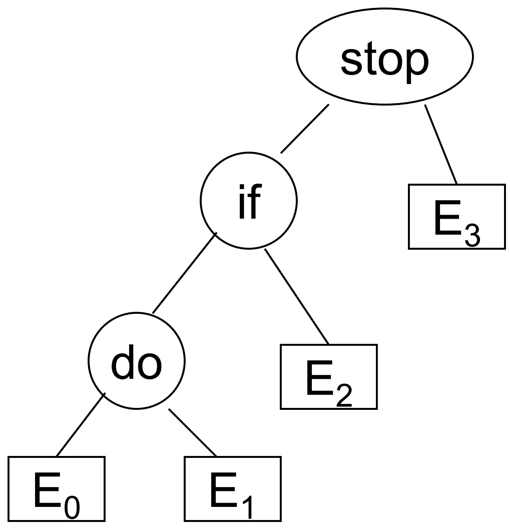


Algorithm search(x)

```
{
        found:=false;
        t:=tree;
        while( (t≠0) and not found ) do
        {
                if( x=t->data )  then found:=true;
                else if( x<t->data ) then t:=t->lchild;
                        else  t:=t->rchild;
        }
       if( not found ) then return 0;
         else  return 1;
}
```

Optimal Binary Search Trees

- Problem
  - Given sequence of identifiers (a_1,a_2 …., $a_{n)}$ with $a_1 < a_2 < \cdots < a_{n.}$
  - Let *p(i) be the* probability *with which we search for* a_i
  - Let q(i) be the probability with which we search for an identifier x such that $a_i < x < a_{i+1}$.
  - Want to build a binary search tree (BST) with minimum expected search cost.



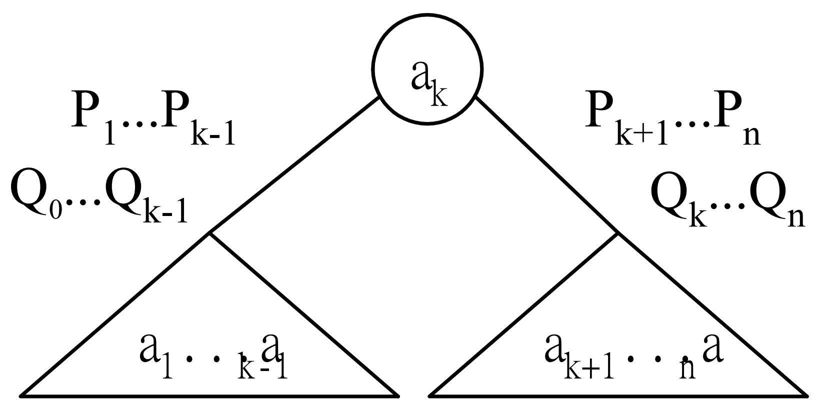
- Identifiers : stop, if, do
  Internal node : successful search, p(i)
- External node : unsuccessful search, q(i)

- The expected cost of a binary tree:

$$\sum_{1 \le i \le n} P_i * \text{level}(a_i) + \sum_{0 \le i \le n} Q_i * (\text{level}(E_i) - 1)$$

The dynamic programming approach

- Make a decision as which of the a_i's should be assigned to the root node of the tree.
- If we choose a_k, then it is clear that the internal nodes for $a_1, a_2, \ldots\ldots, a_{k-1}$ as well as the external nodes for the classes E_0, $E_1, \ldots., E_{k-1}$ will lie in the left subtree *l* of the root. The remaining nodes will be in the right subtree r.



$$\text{cost}(l) = \sum_{1 \le i < k} p(i) * \text{level}(a_i) + \sum_{0 \le i < k} q(i) * (\text{level}(E_i) - 1)$$

$$\text{cost}(r) = \sum_{k < i \le n} p(i) * \text{level}(a_i) + \sum_{k < i \le n} q(i) * (\text{level}(E_i) - 1)$$

- In both the cases the level is measured by considering the root of the respective subtree to be at level 1.

- Using $w(i, j)$ to represent the sum $q(i) + \sum_{l=i+1}^{j} (q(l) + p(l))$, we obtain the following as the expected cost of the above search tree.

$$p(k) + \text{cost}(l) + \text{cost}(r) + w(0, k-1) + w(k, n)$$

- If we use $c(i,j)$ to represent the cost of an optimal binary search tree t_{ij} containing $a_{i+1},\ldots\ldots,a_j$ and $E_i,\ldots,E_j$, then
  cost(l)=$c(0,k-1)$, and cost(r)=$c(k,n)$.

- For the tree to be optimal, we must choose k such that
  $p(k) + c(0,k-1) + c(k,n) + w(0,k-1) + w(k,n)$ is minimum.

Hence, for c(0,n) we obtain

$$c(0,n) = \min_{1 \le k \le n} \left\{ c(0,k-1) + c(k, n) + p(k) + w(0,k-1) + w(k,n) \right\}$$

We can generalize the above formula for any $c(i,j)$ as shown below

$$c(i, j) = \min_{i < k \le j} \left\{ c(i,k-1) + c(k,j) + \underbrace{p(k) + w(i,k-1) + w(k,j)} \right\}$$

$$c(i, j) = \min_{i<k\le j} \left\{ cost(i,k-1) + cost(k,j) \right\} + w(i, j)$$

- Therefore, $c(0,n)$ can be solved by first computing all $c(i, j)$ such that $j - i=1$, next we compute all $c(i,j)$ such that $j - i=2$, then all $c(i, j)$ with $j – i=3$, and so on.

- During this computation we record the root $r(i, j)$ of each tree t_{ij}, then an optimal binary search tree can be constructed from these $r(i, j)$.

- $r(i, j)$ is the value of k that minimizes the cost value.

Note:1. $c(i,i) = 0$, $w(i, i) = q(i)$, and $r(i, i) = 0$ for all $0 \le i \le n$

2. $w(i, j) = p(j) + q(j) + w(i, j-1)$

Ex 1: Let $n=4$, and $(a_1, a_2, a_3, a_4) = (\text{do}, \text{if}, \text{int}, \text{while})$.

Let $p(1:4) = (3, 3, 1, 1)$ and $q(0:4) = (2, 3, 1, 1, 1)$.

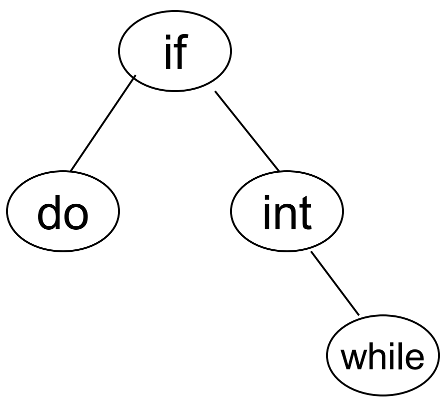
p's and q's have been multiplied by 16 for convenience.

Then, we get

j-i					
0	$w_{00}=2$ $c_{00}=0$ $r_{00}=0$	$w_{11}=3$ $c_{11}=0$ $r_{11}=0$	$w_{22}=1$ $c_{22}=0$ $r_{22}=0$	$w_{33}=1$ $c_{33}=0$ $r_{33}=0$	$w_{44}=1$ $c_{44}=0$ $r_{44}=0$
1	$w_{01}=8$ $c_{01}=8$ $r_{01}=1$	$w_{12}=7$ $c_{12}=7$ $r_{12}=2$	$w_{23}=3$ $c_{23}=3$ $r_{23}=3$	$w_{34}=3$ $c_{34}=3$ $r_{34}=4$	
2	$w_{02}=12$ $c_{02}=19$ $r_{02}=1$	$w_{13}=9$ $c_{13}=12$ $r_{13}=2$	$w_{24}=5$ $c_{24}=8$ $r_{24}=3$		
3	$w_{03}=14$ $c_{03}=25$ $r_{03}=2$	$w_{14}=11$ $c_{14}=19$ $r_{14}=2$			
4	$w_{04}=16$ $c_{04}=32$ $r_{04}=2$				

Computation of c(0,4), w(0,4), and r(0,4)

- From the table we can see that *$c(0,4)=32$* is the minimum cost of a binary search tree for *(a1, a2, a3, a4).*
- The root of tree t_{04} is a_{2}.
- The left subtree is t_{01} and the right subtree t_{24}.
- Tree t_{01} has root a_1; its left subtree is t_{00} and right subtree t_{11}.
- Tree t_{24} has root a_3; its left subtree is t_{22} and right subtree t_{34}.
- Thus we can construct *OBST*.



Ex 2: Let n=4, and (a1,a2,a3,a4) = (count, float, int,while).

Let p(1 : 4) =(1/20, 1/5, 1/10, 1/20) and

q(0: 4) = (1/5,1/10, 1/5,1/20,1/20).

Using the r(i, j)'s construct an optimal binary search tree.

Time complexity of above procedure to evaluate the *c's* and *r's*

- Above procedure requires to compute *c(i, j)* for
  (j - i) = 1,2,……..,n .

- When *j – i = m*, there are *n-m+1 c(i, j)'s* to compute.

- The computation of each of these *c(i, j)'s* requires to find *m* quantities.

- Hence, each such *c(i, j)* can be computed in time *o(m).*

- The total time for all *c(i,j)'s* with *j – i= m* is

$$= m(n-m+1)$$
$$= mn-m^2+m$$
$$= O(mn-m^2)$$

- Therefore, the *total time* to evaluate all the *c(i, j)'s* and *r(i, j)'s* is

$$\sum_{1 \le m \le n} (mn - m^2) = O(n^3)$$

- We can reduce the *time complexity* by using the observation of *D.E. Knuth*

- *Observation:*
  - The optimal k can be found by limiting the search to the range $r(i, j-1) \le k \le r(i+1, j)$

- In this case the *computing* time is $O(n^2)$.

OBST Algorithm

Algorithm OBST(p,q,n)

```
{
    for i:= 0 to n-1  do
    {                                        //  initialize.
            w[ i, i ] :=q[ i ]; r[ i, i ] :=0;  c[ i, i ]=0;
                                             //  Optimal trees  with one node.
            w[ i, i+1 ]:= p[ i+1 ] + q[ i+1 ] + q[ i ] ;
            c[ i, i+1 ]:= p[ i+1 ] + q[ i+1 ] + q[ i ] ;
            r[ i, i+1 ]:= i + 1;
    }
    w[n, n] :=q[ n ]; r[ n, n ] :=0;  c[ n, n ]=0;
```

// Find optimal trees with m nodes.

```
for m:= 2 to n do
{
        for i := 0 to n – m do
        {
                j:= i + m ;
                w[ i, j ]:= p[ j ] + q[ j ] + w[ i, j -1 ];

                                        // Solve using Knuth's result
                x := Find( c, r, i, j );

                c[ i, j ] := w[ i, j ] + c[ i, x -1 ] + c[ x, j ];
                r[ i, j ] :=x;
        }
}
```

```
Algorithm  Find( c, r, i, j )
{
        for k := r[ i, j -1 ] to r[ i+1, j ] do
        {       min :=∞;
                if ( c[ i, k -1 ]  +c[ k, j ] < min ) then
                {
                        min := c[ i, k-1 ] + c[ k, j ];  y:= k;
                }
        }
 return y;
}
```

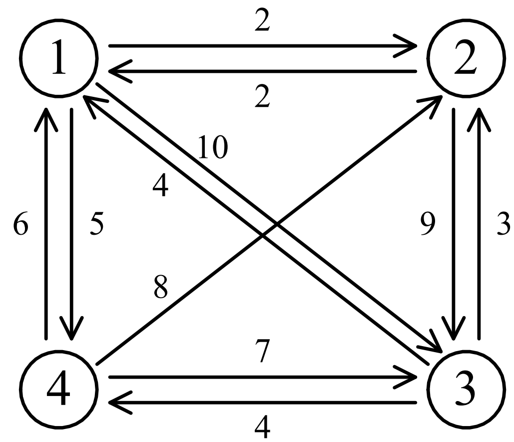
Traveling Salesperson Problem (TSP)

Problem:-

- You are given a set of *n cities.*
- You are given the *distances* between the cities.
- You *start and terminate* your tour at your *home city.*
- You must visit each other city *exactly once*.
- Your mission is to *determine* the *shortest tour*. OR

  minimize the *total distance* traveled.

- e.g. a directed graph :



- Cost matrix:

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 2 & 10 & 5 \\ 2 & 0 & 9 & \infty \\ 4 & 3 & 0 & 4 \\ 6 & 8 & 7 & 0 \end{bmatrix} \end{array}$$

The dynamic programming approach

- 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.

- The *length* of an optimal tour :

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

- The general form:

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

- 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 *equation 2* .

Clearly,

$$g(i, \varnothing) = C_{i1}, \quad 1 \leq i \leq n.$$

- Hence we can use *eq 2* to obtain *g(i, S)* for all *S* of *size 1*. Then we can obtain *g(i, s)* for all *S* of *size 2* and so on.

Thus,

$g(2, \emptyset)=C_{21}=2$, $g(3, \emptyset)=C_{31}=4$

$g(4, \emptyset)=C_{41}=6$

We can obtain

$g(2, \{3\})=C_{23} + g(3, \emptyset)=9+4=13$

$g(2, \{4\})=C24 + g(4, \emptyset)=\infty$

$g(3, \{2\})=C32 + g(2, \emptyset)=3+2=5$

$g(3, \{4\})=C34 + g(4, \emptyset)=4+6=10$

$g(4, \{2\}) = C42 + g(2, \emptyset) = 8+2 = 10$

$g(4, \{3\}) = C43 + g(3, \emptyset) = 7+4 = 11$

Next, we compute $g(i,S)$ with $|S| = 2$,

$g(2,\{3,4\}) = \min\{c23+g(3,\{4\}), c24+g(4,\{3\})\}$

$= \min\{19, \infty\} = 19$

$g(3,\{2,4\}) = \min\{c32+g(2,\{4\}), c34+g(4,\{2\})\}$

$= \min\{\infty, 14\} = 14$

$g(4,\{2,3\}) = \min\{c42+g(2,\{3\}), c43+g(3,\{2\})\}$

$= \min\{21, 12\} = 12$

Finally,

We obtain

$$
\begin{aligned}
g(1,\{2,3,4\}) &= \min\{\ c12+g(2,\{3,4\}),\\
&\qquad c13+g(3,\{2,4\}),\\
&\qquad c14+g(4,\{2,3\})\ \}\\
&= \min\{2+19,10+14,5+12\}\\
&= \min\{21,24,17\}\\
&= 17.
\end{aligned}
$$

- A tour can be constructed if we retain with each *g(i, s)* the value of *j* that *minimizes* the tour distance.
- Let *J(i, s)* be this value, then *J(1, { 2, 3, 4 }) = 4*.
- Thus the tour starts from *1* and goes to *4.*

- The remaining tour can be obtained from *g(4, { 2,3 }).* So *J(4, { 3, 2 })=3*

- Thus the next edge is *<4, 3>.* The remaining tour is *g(3, { 2 }).* So *J(3,{ 2 })=2*

The optimal tour is: (1, 4, 3, 2, 1)

Tour distance is 5+7+3+2 = 17

All pairs shortest path problem

Floyd-Warshall Algorithm

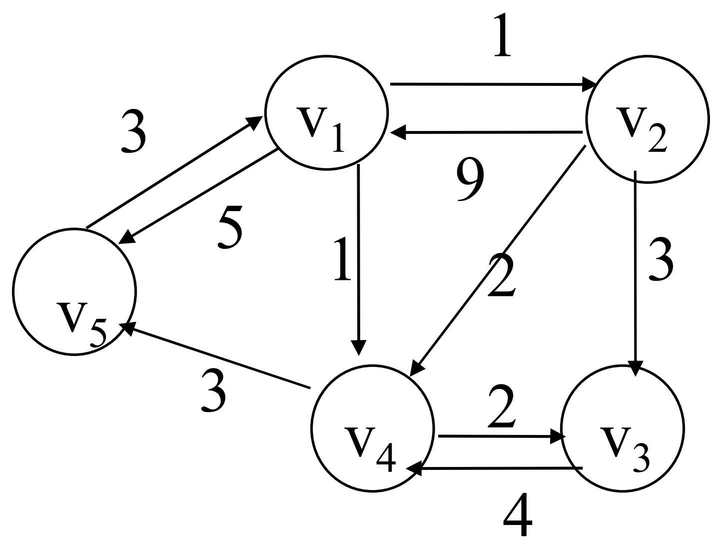
All-Pairs Shortest Path Problem

- Let G=(V,E) be a *directed* graph consisting of n vertices.
- *Weight* is associated with each edge.

- The problem is to *find a shortest path* between *every pair of nodes.*

Ex:-

$$
\begin{array}{c|ccccc}
 & 1 & 2 & 3 & 4 & 5 \\
\hline
1 & 0 & 1 & \infty & 1 & 5 \\
2 & 9 & 0 & 3 & 2 & \infty \\
3 & \infty & \infty & 0 & 4 & \infty \\
4 & \infty & \infty & 2 & 0 & 3 \\
5 & 3 & \infty & \infty & \infty & 0
\end{array}
$$



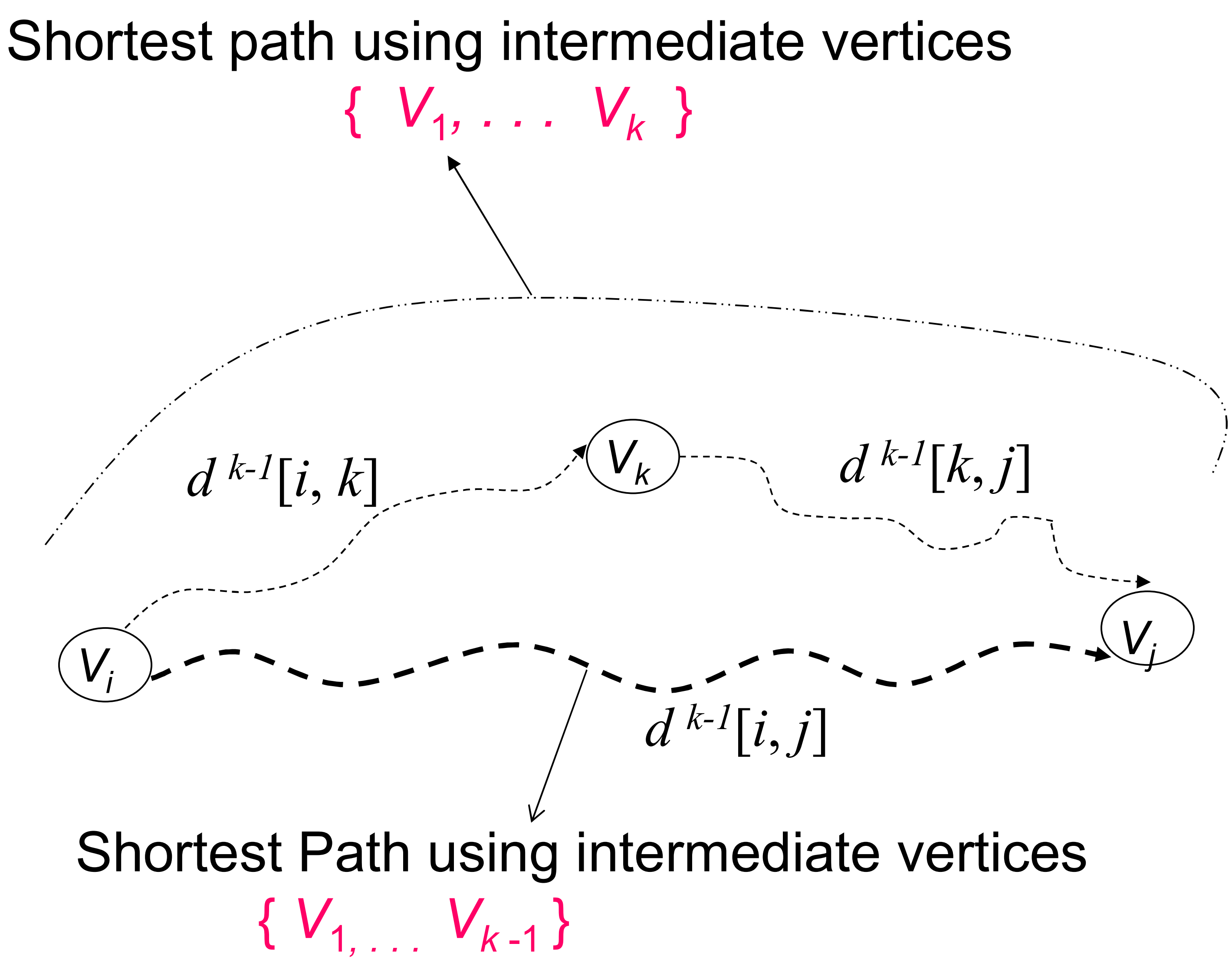
Idea of Floyd-Warshall Algorithm

- Assume vertices are $\{1,2,\ldots\ldots n\}$
- Let $d^{\,k}(i, j)$ be the length of a shortest path from i to j with intermediate vertices numbered not higher than k where $0 \le k \le n$, then
- $d^0(i, j) = c(i, j)$ (no intermediate vertices at all)
- $d^{\,k}(i, j) = \min\{ d^{k-1}(i, j),\ d^{k-1}(i, k) + d^{k-1}(k, j) \}$
- $d^{\,n}(i, j)$ is the *length* of a shortest path from *i to j*

- In summary, we need to find d^{n} with d^{0} =cost matrix .

- General formula

$$d^{k}[i, j] = \min \left\{ d^{k-1}[i, j], d^{k-1}[i, k] + d^{k-1}[k, j] \right\}$$



Shortest path using intermediate vertices
{ V_1, . . . V_k }
$d^{k-1}[i, k]$
V_k
$d^{k-1}[k, j]$
V_i
V_j
$d^{k-1}[i, j]$
Shortest Path using intermediate vertices
{ V_{1}, . . . V_{k-1} }

$$d^0 = \begin{bmatrix} 0 & 1 & \infty & 1 & 5 \\ 9 & 0 & 3 & 2 & \infty \\ \infty & \infty & 0 & 4 & \infty \\ \infty & \infty & 2 & 0 & 3 \\ 3 & \infty & \infty & \infty & 0 \end{bmatrix}$$

d^1 =

d^2 =

d^3 =

d^4 =

d^5 =

Algorithm

Algorithm AllPaths(c, d, n)

```
// c[1:n,1:n] cost matrix
// d[i,j] is the length of a shortest path from i to j
{
          for i := 1 to n do
             for j := 1 to n do
                    d [ i, j ] := c [ i, j ] ;     // copy c into d

          for k := 1 to n do
             for i := 1 to n do
                    for j := 1 to n do
                              d [ i, j ] := min ( d [ i, j ] , d [ i, k ] + d [ k, j ] );
}
```

Time Complexity is $O(n^3)$

0/1 Knapsack Problem

Let $x_i = 1$ when item i is selected and let $x_i = 0$ when item i is not selected.

$$\text{maximize} \quad \sum_{i=1}^{n} p_i x_i$$

$$\text{subject to} \quad \sum_{i=1}^{n} w_i x_i <= c$$

and $x_i = 0$ or 1 for all i

All profits and weights are positive.

Sequence Of Decisions

- Decide the x_i values in the order $x_1, x_2, x_3, \ldots, x_n$.

OR

- Decide the x_i values in the order $x_n, x_{n-1}, x_{n-2}, \ldots, x_1$.

Problem State

- Suppose that decisions are made in the order $x_1, x_2, x_3, \ldots, x_n$.
- The initial state of the problem is described by the pair (1, m).
  - Items 1 through n are available
  - The available knapsack capacity is m.
- Following the first decision the state becomes one of the following:
  - (2, m) … when the decision is to set $x_1 = 0$.
  - (2, m-w_1) … when the decision is to set $x_1 = 1$.

Problem State

- Suppose that decisions are made in the order $x_n, x_{n-1}, x_{n-2}, \ldots, x_1$.
- The initial state of the problem is described by the pair (n, m).
  - Items 1 through n are available
  - The available knapsack capacity is m.
- Following the first decision the state becomes one of the following:
  - $(n-1, m)$ … when the decision is to set $x_n = 0$.
  - $(n-1, m-w_n)$ … when the decision is to set $x_n = 1$.

Dynamic programming approach

- Let $f_n(m)$ be the value of an optimal solution, then

$$f_n(m) = \max \{ f_{n-1}(m),\ f_{n-1}(m-w_n) + p_n \}$$

General formula

$$f_i(y) = \max \{ f_{i-1}(y),\ f_{i-1}(y-w_i) + p_i \}$$

- We use set s^i is a pair (P, W)

  where $P= f_i(y)$, $W=y$

- Note That $s^0 =(0, 0)$

- We can compute s^{i+1} from s^i by first computing

$$s^i_1 =\{ (P, W) / (P- p_{i+1}, W- w_{i+1}) \text{€}\ s^i \}$$

OR

$S^i_1 = S^i + (p_{i+1}, w_{i+1})$

Merging :- s^{i+1} can be computed by merging the pairs in s^i and s^i_1

Purging :- if s^{i+1} contains two pairs (p_j, w_j) and (p_k, w_k) with the property that $p_j \leq p_k$ and $w_j \geq w_k$ then the pair (p_j, w_j) can be discarded.

- When generating s^i's, we can also purge all pairs *(p, w)* with $w > m$ as these pairs determine the value of $f_n(x)$ only for $x > m$.
- The optimal solution $f_n(m)$ is given by the *highest profit pair (last pair in* s^n *) .*

Set of 0/1 values for the x_i's

- *Set of 0/1 values for x_i's can be determined by a search through the s^i s*
  - Let (p, w) be the highest profit tuple in s^n

Step1: if $(p, w) \in s^n$ *and* $(p, w) \notin s^{n-1}$

$x_n = 1$

otherwise $x_n = 0$

This leaves us to determine how either (p, w) or $(p - p_n, w - w_n)$ *was obtained in* S^{n-1} .

This can be done recursively (Repeat Step1).

Ex: knapsack instance n=3, $(w_1, w_2, w_3)=(2,3,4)$, $(p_1,p_2,p_3)=(1,2,5)$, and m=6. for this data we have

S^0 ={ (0,0) } ; S_1^0 ={ (1,2) }

S^1 ={ (0,0), (1,2) } ; S_1^1={ (2,3), (3,5) }

S^2 ={ (0,0), (1,2), (2,3), (3,5) } ; S_1^2={ (5,4), (6,6), (7,7), (8,9) }

S^3 ={ (0,0), (1,2), (2,3), (5,4), (6,6) }

Note that the pair (3, 5) has been eliminated from s^3 as a result of the purging rule.

Also, note that the pairs (7, 7) and (8, 8) have been eliminated from s^3. Because, for these pairs w>m i.e., 7>6 and 9>6.

Therefore, the optimal solution is (6,6) (highest profit pair).

If (p, w) is the highest profit pair in sn, a set of 0/1 values for the x_i's can be determined by a search through the s^i 's.

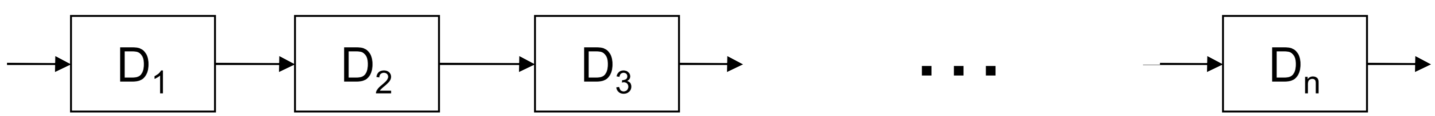
We can set $x_n=0$ if (p, w) € s^{n-1} else $x_n=1$. This leaves us to determine how

either (p, w) or $(p - p_n, w - w_n)$ was obtained in s^{n-1}. This can be done recursively.

Solution vector is $(x_1, x_2, x_3)=(1, 0, 1)$.

Reliability Design

- The problem is to design a system that is composed of *several devices* connected in *series*.



n devices connected in series

- Let r_i be the reliability of device D_i (that is, r_i is the probability that device i will function properly).

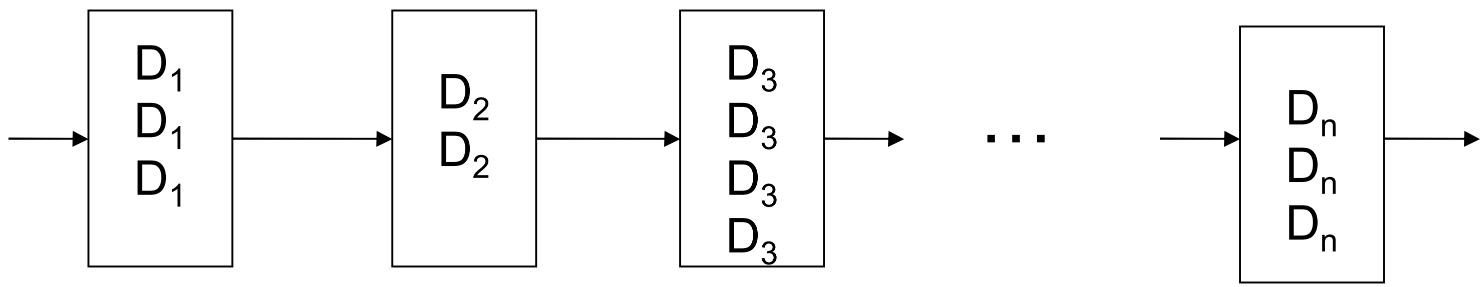
- Then, the reliability of *entire system* is $\pi\, r_i$

- Even if the *individual* devices are very reliable, the reliability of the entire system may *not be very* good.

- Ex. If $n=10$ and $r_i = 0.99$, $1 \le i \le 10$, then $\pi\, r_i = 0.904$

- *Hence, it is desirable to duplicate devices.*

- Multiple copies of the *same device type* are connected in parallel as shown below.



Multiple devices connected in parallel in each stage

- If stage ***i*** contains m_i copies of device D_i, *then the probability that all* m_i *have malfunction is* $(1-r_i)^{m_i}$. *Hence the reliability of stage i becomes* $1-(1-r_i)^{m_i}$.

Ex:- If $r_i = .99$ *and* $m_i = 2$ *, the stage reliability becomes* 0.9999

- *Let* $\Phi_i(m_i)$ *be the reliability of stage* i, $i \le n$
- *Then, the reliability of system of n stages is* $\prod_{1 \le i \le n} \Phi_i(m_i)$

- Our problem is to use *device duplication* to *maximize reliability*. This maximization is to be carried out *under a cost constraint.*
- *Let c_i be the cost of each device i and c be the maximum allowable cost of the system being designed.*
- *We wish to solve the following maximization problem:*

$$\text{maximize} \quad \prod_{1 \le i \le n} \Phi_i(m_i)$$

$$\text{subjected to} \quad \sum_{1 \le i \le n} c_i m_i \le c$$

$$m_i \ge 1 \text{ and integer}, \quad 1 \le i \le n$$

Dynamic programming approach

- Since, each $c_i > 0$, each m_i must be in the range $1 \le m_i \le u_i$, where

$$u_i = \left\lfloor \left(c + c_i - \sum_{1 \le j \le n} c_j \right) \Big/ c_i \right\rfloor$$

- The upper bound u_i follows from the observation that $m_i \ge 1$.
- The optimal solution $m_1, m_2, \ldots\ldots, m_n$ is the result of a sequence of decisions, one decision for each m_i

- Let $f_n(c)$ be the reliability of an optimal solution, then

$$f_n(c) = \max_{1 \le m_n \le u_n} \{ \Phi_n(m_n) \ f_{n-1}(c - c_n m_n) \}$$

General formula

$$f_i(x) = \max_{1 \le m_i \le u_i} \{ \Phi_i(m_i) \ f_{i-1}(x - c_i m_i) \}$$

- Clearly, $f_0(x)=1$, for all x, $0 \le x \le c$

- Let s^i consist of tuples of the form (f, x)

Where $f = f_i(x)$

Purging rule :- if s^{i+1} contains two pairs (f_j, x_j) and (f_k, x_k) with the property that $f_j \le f_k$ and $x_j \ge w_k$, then we can purge (f_j, x_j)

- When generating s^{i}'s, we can also purge all pairs (f, x) with $c - x < \sum_{i+1 \le k \le n} c_k$ as such pairs will not leave sufficient funds to complete the system.

- The optimal solution $f_n(c)$ is given by the *highest reliability pair.*

- *Start with* $S^0 = (1, 0)$

Ex: (D1, D2, D3)=(30, 15, 20), c=105, r1=.9, r2=.8, r3=.5.

Therefore, u1=(105+30-65)/30=2.33=2

u2=(105+15-65)/15=3.66=3

u3=(105+20-65)/20=3

S^0={ (1,0) } $s_1 0$={ (.9, 30) }, $s_2 0$ ={ (.99, 60) } merge $s_1 0$ and $s_2 0$ to get s^1

S^1={ ((.9, 30), (.99, 60) } $s_1 1$={ (.72, 45), (.792, 75) } $s_2 1$={ (.864, 60), (.9504, 90) } $s_3 1$ ={ (.8928, 75), (.98208, 105) } merge $s_1 1$, $s_2 1$, and $s_3 1$ to get s^2

S^2={ (.72, 45), (.864, 60), (.8928, 75) } Note 1: Tuple (.792, 75) is eliminated –purging rule.

2: Tuples (.9504,90) & (.98208,105) will not leave sufficient funds to complete the system.

$s_1 2$ ={ (.36, 65),(.432, 80),~~(.4464, 95)~~ } purging rule

$s_2 2$={ (.54, 85),(.648, 100),~~(.6696, 115)~~ } Total cost is >105

$s_3 2$={ ~~(.63, 105)~~,~~(.756, 120)~~,~~(.7812, 135)~~ }

S^3={ (.36, 65), (.432, 80), (.54, 85), (.648, 100) ----- **The best design has a reliability of .648 and a cost of 100.**

Tracing back through the si's , we can determine that $m_1=1, m_2=2, m_3=2$.