UNIT – III

Unit-III/Lecture-01

Concept of Dynamic Programming

Concept of dynamic programming:

Dynamic Programming(usually referred to as **DP**) is a powerful technique that allows to solve many different types of problems in time O(n2) or O(n3) for which a naive approach would take exponential time. In the word dynamic programming the word programming stands for "planning" and it does not mean by computer programming. Dynamic programming is typically applied to optimization problem.

Dynamic Programming is an algorithmic paradigm that solves a given complex problem by breaking it into subproblems and stores the results of subproblems to avoid computing the same results again. Following are the two main properties of a problem that suggest that the given problem can be solved using Dynamic programming.

1) Overlapping Subproblems
2) Optimal Substructure

1) Overlapping Subproblems:
Like Divide and Conquer, Dynamic Programming combines solutions to sub-problems. Dynamic Programming is mainly used when solutions of same subproblems are needed again and again. In dynamic programming, computed solutions to subproblems are stored in a table so that these don't have to recomputed. So Dynamic Programming is not useful when there are no common (overlapping) subproblems because there is no point storing the solutions if they are not needed again. For example, Binary Search doesn't have common subproblems. If we take example of following recursive program for Fibonacci Numbers, there are many subproblems which are solved again and again.

```
int fib(int n)
{
  if ( n <= 1 )
    return n;
  return fib(n-1) + fib(n-2);
}
```

2) Optimal Substructure
A problem is said to have **optimal substructure** if an optimal solution can be constructed efficiently from optimal solutions of its subproblems. This property is used to determine the usefulness of dynamic programming and greedy algorithms for a problem.

There are two ways of doing this.
1) Top-Down : Start solving the given problem by breaking it down. If you see that the problem has been solved already, then just return the saved answer. If it has not been solved, solve it and save the answer. This is usually easy to think of and very intuitive. This is referred to as ***Memoization***.

2) Bottom-Up : Analyze the problem and see the order in which the sub-problems are solved and start solving from the trivial subproblem , up towards the given problem. In this process, it is guaranteed that the subproblems are solved before solving the problem. This is referred to as

Dynamic Programming:

Note that divide and conquer is slightly a different technique. In that, we divide the problem in to non-overlapping subproblems and solve them independently, like in mergesort and quick sort.

Principal of optimality: [RGPV June-2014(2)]

The principle of optimality states that no matter what the first decision, the remaining decisions must be optimal with respect to the state that results from this first decision.

This principle implies that an optimal decision sequence is comprised for some formulations of some problem.

Since the principle of optimality may not hold for some formulations of some problems, it is necessary to verify that it does not hold for the problem being solved.

Dynamic programming cannot be applied when this principle does not hold.

S.NO	RGPV QUESTIONS	Year	Marks
Q.1			
Q.2			

Unit-III/Lecture-02

0/1 knapsack Problem

0/1 knapsack: [RGPV June-2014(3)]

Given weights and values of n items, put these items in a knapsack of capacity W to get the maximum total value in the knapsack. In other words, given two integer arrays val[0..n-1] and wt[0..n-1] which represent values and weights associated with n items respectively. Also given an integer W which represents knapsack capacity, find out the maximum value subset of val[] such that sum of the weights of this subset is smaller than or equal to W. You cannot break an item, either pick the complete item, or don't pick it (0-1 property).

Problem Description

If we are given n objects and a knapsack or a bag in which the object i that has weight w_i is to be placed. The knapsack has a capacity W. Then the profit that can be earned is $p_i x_i$. The objective is to obtain filling of knapsack with maximum profit earned.Maximized $p_i x_i$. subject to constraint $w_i x_i$<=W Where 1<=i<=n and n is total no. of objects and x_i =0 or 1.

Steps and Notations

Step-1:

Let $f_i(y_i)$ be the value of optimal solution. Then s^i is pair (p,w) where p=f(y_j) and w=y_j Initially
$s^0 = \{(0,0)\}$
We can compute s^{i+1} from s^i **.** These computations of s^i are basically the sequence of decisions made for obtaining the optimal solutions.

Step-2:

We can generate the sequence of decisions in order to obtain the optimum selection for solving the knapsack problem.

Let x_n be the optimum sequence. Then there are two instances $\{x_n\}$ and $\{x_{n-1},x_{n-2}\ldots x_1\}$.
So from $\{x_{n-1},x_{n-2}\ldots x_1\}$ we will choose the optimum sequence with respect to x_n.
The selection of sequence from remaining set should be such that we should be able to fulfill the condition of filling knapsack capacity W with maximum profit.
Otherwise $\{x_{n-1},x_{n-2}\ldots x_1\}$ is not optimum.
This proves that 0/1 knapsack problem is solved using principle of optimality.

Step-3:

Let $f_i(y_j)$ be the value of optimal solution. Then fi(y) = max$\{f_{i-1}(y), f_{i-1}(y-w_i)+p_i\}$
Initially compute

$$s^0=\{(0,0)\}$$

$s^i_1 = \{(P,W)|(P-p_i, W-w_i)$ belongs to $s^i\}$

S^{i+1} can be computed by merging s^i and s^i_1

Purging rule

If s^{i+1} contains (P_j, W_j) and (P_k, W_k) these two pairs such that Pj<=Pk and Wj>=Wk, then (P_j,W_j) can be eliminated . This purging rule is also called as dominance rule. In purging rule basically the dominated tuples gets purged. In short remove the pair with less profit and more weight.

Problem-1

Solve knapsack instance M=8, and n=4. let p_i and w_i are as shown below.

i	p_i	w_i
1	1	2
2	2	3
3	5	4
4	6	5

Solution-

s^0={(0,0)} initially

s^0_1={(1,2)}

That means while building s^0_1 we select the next i^{th} pair.

For s^0_1 we have selected first (P,W) pair which (1,2).

S^1 ={merge s^0 and s^0_1}

={(0,0),(1,2)}

s^1_1={select next (P,W) pair and add it with s^1}

={(2,3),(2+0,3+0), (2+1,3+2)}

s^1_1 ={(2,3),(3,5)} // repetition of (2,3) avoided.

S^2 ={merge candidates from s^1 and s^1_1}

={ (0,0), (1,2), (2,3), (3,5)}

S^2_1 ={select next (P,W) pair and add it with s^2}

={(5,4),(6,6),(7,7),(8,9)}

S^3 ={merge candidates from s^2 and s^2_1}

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

Note that the pair (3, 5) is purged form s^3.

Because let us assume (P_j, W_j) = (3, 5) and (P_k, W_k) = (5, 4).

Here P_j<=P_k and W_j>W_k is true hence we will eliminate pair (P_j, W_j) = (3, 5) from s^3.

S^3_1 ={select next (P,W) pair and add it with s^3}

S^3_1 ={(6,5),(7,7),(8,8),(11,9),(12,11) ,(13,12) ,(14,14)}

S^4={(0,0),(1,2),(2,3) ,(5,4) ,(6,6) ,(7,7),(8,9) ,(6,5) ,(8,8),(11,9),(12,11) ,(13,12) ,(14,14)}

Now we are interested in M=8. We get pair (8, 8) in s^4.

Hence we will set x_4=1.

Now we select next object (P-p_4) and (W-w_4).

i.e (8-6) and (8-5).

i.e (2,3)

Pair (2,3) belongs to s^2.hence set x_2=1.

So we get the final solution as (0, 1, 0, 1).

S.NO	RGPV QUESTIONS	Year	Marks
Q.1			
Q.2			

Unit-III/Lecture-03

Multistage graph

Multistage graph:[RGPV June-2014(2)]

A multistage graph G=(V,E) is a directed graph in which the vertices are partitioned into $k \geq 2$ disjoint sets V_i, $1 \leq i \leq k$. In addition if (u, v) is an edge in E, then u ε V_i and v ε V_{i+1}, for some i, $1 \leq i \leq k$. The sets V_1 and V_k are such that $|V_1|=|V_k|=1$.
Let s and t, respectively be the vertex in V_1 and V_k. The vertex s is the source, and t the sink.

The above definition says that the vertices are divided into several disjoint partitions in a multistage graph. Each partition is called as a stage which contains several vertices. The first and last partition /stage of the graph contains one vertex each, namely, the source (s) and the sink (t).

Let c(i, j) be the cost of edge (i, j). The cost of a path from s to t is the sum of the costs of the edges on the path. The multistage graph problem is to find a minimum cost path from s to t. It should be noted that each set V_i defines a stage in the graph. Because of the constraints on E (the set of edges), every path from s to t starts from the source vertex in stage 1, goes to stage 2, then to stage 3 and so on, and eventually terminates at the sink vertex in stage K, i.e. the last stage. Consider the directed graph given below.

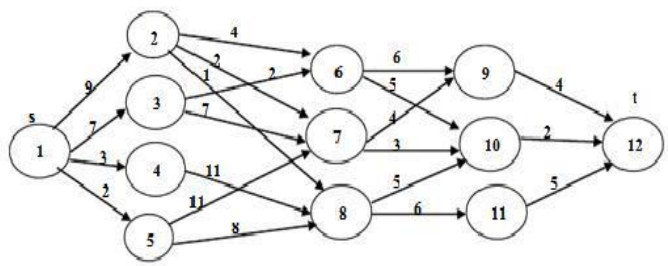


Fig-3.1 Multistage Graph with Five Stages

There are five stages in the graph i.e. k = 5 in this graph. The five stages are listed below:
Stage 1: V_1 = {1}, the source vertex s.
Stage 2: V_2 = {2, 3, 4, 5}
Stage 3: V_3 = {6,7,8}
Stage 4: V_4 = {9,10, 11}
Stage 5: V_5 = {12}, the sink vertex t.

From the graph given in Fig-3.1, it can be noticed that the shortest path from the source vertex to sink vertex is "1 - 2 -7 - 10 -12". Tthe path from s to t starts from the source vertex in stage 1, goes to stage 2, then to stage 3 and so on, and terminates at the sink vertex in stage 5. i.e. the shortest path from s to t starts from the source vertex, 1, which is in stage1, travels through vertex 2 which is in stage 2, vertex 7 which is in stage 3, vertex 10 which is in stage 4 and terminates at the sink vertex, 12, which is in stage 5. There are many real-life problems that can be formulated as multistage graph problem. Few examples include resource allocation for a project in a software company or manufacturing company, project management, job scheduling in operating system etc.

Finding the Shortest Path from Source to Sink

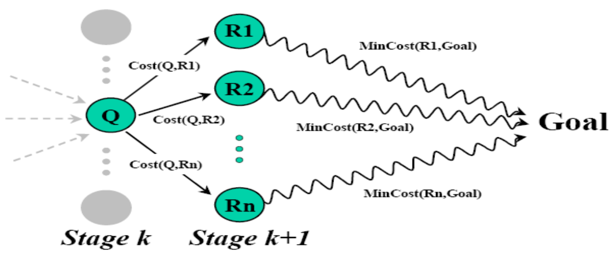
A dynamic programming formulation for k-stage graph problem is obtained by first noticing that every s to t path is the result of a sequence of (k-2) decisions. The i^{th} decision involves determining which vertex in V_{i+1}, 1< i < k-2, is to be on the path. It is easy to see that the principle of optimality holds.

From the weights assigned in the graph, it is easy to observe the following values:

cost (1, 2) = 9, cost (1, 3) =7, cost (1, 4)= 3 and cost (1,5) = 2.
cost (2, 6) = 4, cost (2, 7) = 2 and cost (2, 8) = 1.
cost (3,6) = 2 and cost (3,7) = 7.
cost (4,8)=11.
cost (5,7)=11 and cost (5,8)=8.
cost (6,9) = 6 and cost (6,9)=5.
cost (7,9) = 4 and cost (7,10) = 3.
cost (8,10) = 5 and cost (8,11)=6.
cost (9,12) = 4.
cost (10,12) = 2.
cost (11,12) = 5.

There are two approaches, namely, forward approach and backward approach, to find the shortest path from the source node to the sink node in a multistage graph.

Forward approach:
Assume at stage k that we know the min cost path from each node in stage k+1 to the goal state.



MinCost(Q,Goal)=Min(Cost(Q,R1)+MinCost(R1,Goal), Cost(Q,R2)+MinCost(R2,Goal), …,
Cost(Q,Rn)+MinCost(Rn,Goal))

The cost is computed as follows using the forward approach.

cost(i , j) = min { c(j , l) + cost(i+1, l) }……………… (1)
$l \; \varepsilon \; V_{i+1}$
< j, l> ε E, more than one vertex is considered for l

Since, cost(k-1, j) = c(j ,t) if <j, t> ε E and cost (k-1, j) = ∞ if <j, t> does not belong to E.

We need to solve for cost (1, s) by first computing cost (k-2, j) for all j ε V_{k-2}, then computing cost (k-3, j) for all j ε V_{k-3} etc. and finally cost (1, s). The computations using the forward approach based on formula (1) for the graph shown in Fig-3.1 .

i= k-2
cost(3,6) = Min { 6 + cost(4,9), 5 + cost(4,10)} = 7
cost(3,7) = Min { 4 + cost(4,9), 3 + cost(4,10)} = 5
cost(3,8) = Min { 5 + cost(4,10), 6 + cost(4,11)} = 7
cost(2,2) = Min { 4 + cost(3, 6), 2 + cost(3,7), 1 + cost(3,8)} = 7
cost(2,3) = Min { 2 + cost(3, 6), 7 + cost(3,7)} = 7
cost(2,4) = Min { 11 + cost(3, 8)} = 18
cost(2,5) = Min { 11 + cost(3, 7), 8 + cost(3,8)} = 15
cost(1,1) = Min { 9 + cost(2, 2), 7 + cost(2,3), 3 + cost(2, 4), 2 + cost(2,5)} =16
suppose D(i,j) = r where r minimize the value of c[j,r] + cost(i+1,r)

It should be noted that in the calculation of cost (2,2), we have reused the values of cost (3, 6) and cost (3, 7) and cost (3, 8), and thereby avoiding the recomputation. A minimum cost path from s to t has the cost of 16. This path can be determined easily if we record the decision made at each state (vertex).

D(3,6) = 10 D(2,3) = 6

D(3,7) = 10 D(2,4) = 8

D(3,8) = 10 D(2,5) = 8

D(2,2) = 7 D(1,1) = 2

1 → V2 → V3 → V4 → 12

V2 = D(1,1) = 2

V3 = D(2,2) = 7

V4 = D(3,7) = 10

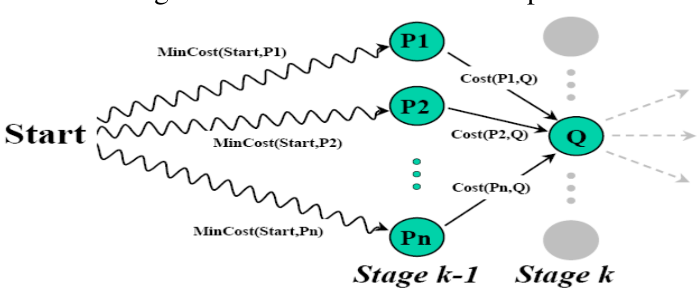
The path = **1 → 2 → 7 → 10 → 12**

Let the minimum-cost path be s = 1, v_2, v_3, ….v_{k-1}, t. It is easy to see that v_2=D(1,1) = 2, v_3 = D(2, D(1,1)) = 7 and v_4 = D (3, D(2, D(1,1))) = D(3,7) = 10. The minimum-cost path from the source node, s, to the sink node, t, for the graph shown in Figure 1 using the forward approach is "1-2-7-10-12". The minimum cost is 16.

```
void FGraph (Graph G, int k, int n, int 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 <i , j>.  p[1 :k] is a minimum – cost path.
{
        float cost[MAXSIZE];
        int d[MAXSIZE],  r;
        cost[n] = 0.0;
        for (int j=n-1; j>=1; j--)  //compute cost [j]
        {
        //Let r be a vertex such that < j , r > 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; j<=k-1; j++)
                p[j] = d[p[j-1]];
}
```

Backward approach:

Assume at stage k that we know the min cost path from start state to each node in stage k - 1.



MinCost(Q,Goal)=Min(Cost(P1, Q)+MinCost(Start, P1), Cost(P2, Q)+MinCost(Start, P2), …, Cost(Pn)+MinCost(Start, Pn))

The backward approach is similar to forward approach. The computation in forward approach start from $V_{(k-2)}$, whereas, in backward approach it starts from V_3. Backward approach employs the following computation. 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).

$$bcost(i, j) = \min_{\substack{l \in V_{i-1} \\ \langle l, j\rangle \in E}} \{ c(i-1, l) + bcost(l, j) \} \ldots\ldots\ldots\ldots\ldots\ldots (2)$$

and more than one vertex is considered for l.

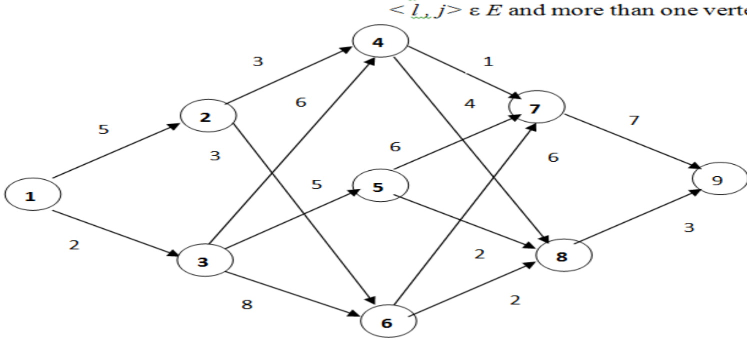


Fig-3.2 Multistage Graph with Five Stages

The computations by employing backward approach using formula (2) are given below:
bcost(i,j) = min{bcost(i-1,l)+c(l,j)}
bcost(3,4) = min{bcost(2,2)+c(2,4),bcost(2,3)+c(3,4)} = min{8,8} = 8
bcost(3,5) = min{bcost(2,3)+c(3,5)} = min{7} = 7
bcost(3,6) = min{bcost(2,2)+c(2,6),bcost(2,3)+c(3,6)} = min{8,10} = 8
bcost(4,7) = min{bcost(3,4)+c(4,7),bcost(3,5)+c(5,7)} = min{9,13} = 9
bcost(4,8) = min{bcost(3,4)+c(4,8),bcost(3,5)+c(5,8),bcost(3,6)+c(6,8)} = min{12,9,10} = 9
bcost(5,9) = min{bcost(4,7)+c(7,9),bcost(4,8)+c(8,9)} = min{16,12} = 12

Computations to find the minimum path are given below:
p[j]=d[p[j+1]], where p[1]=1, p[k]=n, for(j=k-1;j>=2;j--)

J:	1	2	3	4	5	6	7	8	9
d[j]:				2	3	2	4	5	8
p[j]:	1	3	5	8	9				

Therefore , the minimum path using the backward approach for the graph given in Figure 3.2 is "1-3-5-8-9" and the minimum cost is 12. If the graph G is represented by adjacency lists, if G has **|E|** edges, then the time complexity of multistage graph problem is O (|V| + |E|).

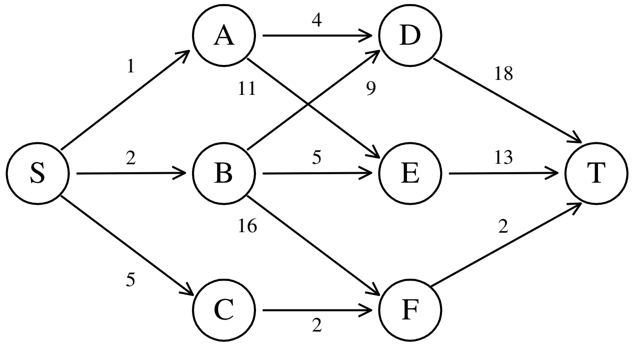
S.NO	RGPV QUESTIONS	Year	Marks
Q.1			
Q.2			

Unit-III/Lecture-04

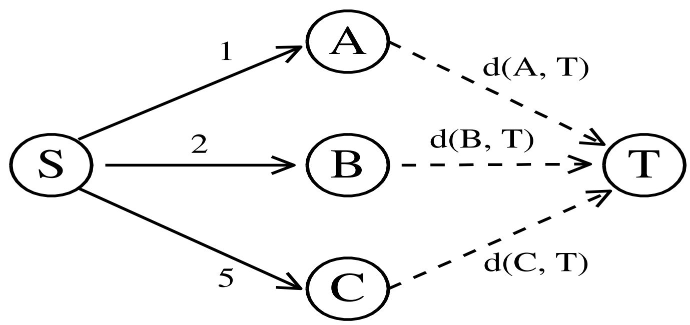
Problems based on Multistage graph

:[RGPV/June-2014(7)]

Q.1 Find a minimum cost path from 'S' to 't' in multistage graph using dynamic programming?

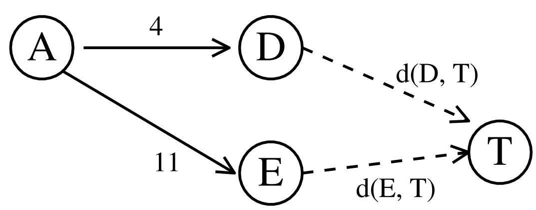


Solution:

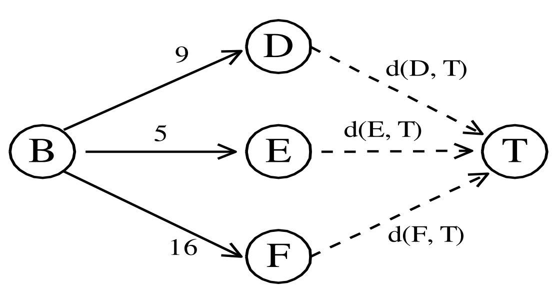


d(S, T) = min{1+d(A, T), 2+d(B, T), 5+d(C, T)}

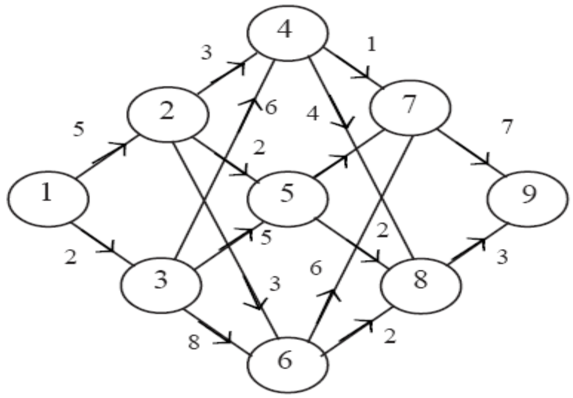
d(A,T) = min{4+d(D,T), 11+d(E,T)}
= min{4+18, 11+13} = 22.



d(B, T) = min{9+d(D, T), 5+d(E, T), 16+d(F, T)}
= min{9+18, 5+13, 16+2} = 18.
d(C, T) = min{ 2+d(F, T) } = 2+2 = 4
d(S, T) = min{1+d(A, T), 2+d(B, T), 5+d(C, T)}
= min{1+22, 2+18, 5+4} = 9.



Q.2 Find a minimum cost path from 'S' to 't' in multistage graph using dynamic programming? [RGPV JUNE-2014]



S.NO	RGPV QUESTIONS	Year	Marks
Q.1			
Q.2			

Unit-III/Lecture-05

Reliability Design

Reliability Design:

Reliability means the ability of an apparatus, machine, or system to consistently perform its intended or required function or mission, on demand and without degradation or failure.

Reliability design using dynamic programming is used to solve a problem with a multiplicative optimization function. The problem is to design a system which is composed of several devices connected in series (below Fig-3.3(b)). Let *r;* be the reliability of device *D;* (i.e. *r;* is the probability that device *i* will function properly). Then, the reliability of the entire system is πr_i. Even if the individual devices are very reliable (the r_i*'s* are very close to one), the reliability of the system may not be very good.

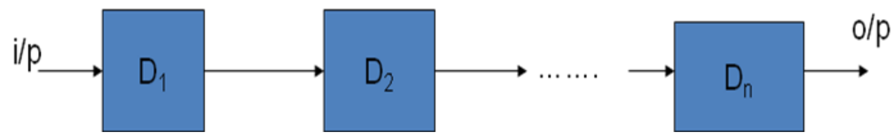


Fig-3.3(a) Devices connected in series

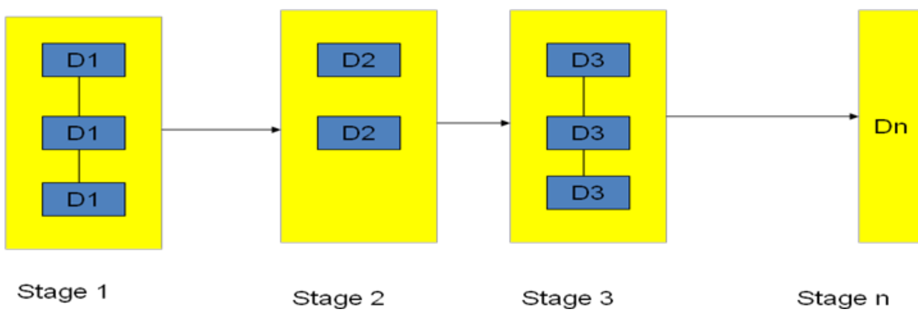


Fig-3.3(b) Multiple Devices Connected in Parallel in each stage

Multiple copies of the same device type are connected in parallel (Fig-3.3(b)) through the use of switching circuits. The switching circuits determine which devices in any given group are functioning properly. They then make use of one such device at each stage.

If stage i contains m_i copies of device D_i then the probability that all m_i have a malfunction is $(1 - r_i)^{mi}$. Hence the reliability of stage i becomes $1 - (1 - r_i)^{m_i}$. Thus, if $r_i = 0.99$ and $m_i = 2$ the stage reliability becomes 0.9999. In any practical situation, the stage reliability will be a little less than $1 - (1 - r_i)^{m_i}$ because the switching circuits themselves are not fully reliable. Also, failures of copies of the same device may not be fully independent (e.g. if failure is due to design defect). Let us assume that the reliability of stage *i* is actually given by a function $\Phi_i(m_i)$, $1 <= i <= n$. (It is quite conceivable that $\Phi_i(m_i)$ may decrease after a certain value of *m* ;). The reliability of the system of stages is $\prod_{1<=i<=n} \Phi_i(m_i)$.

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 unit of device *i* and let *c* be the maximum allowable cost of the system being designed.
We wish to solve the following maximization problem:

maximize $\prod_{1<=i<=n} \Phi_i(m_i)$

subject to $\sum_{1<=i<=n} c_i m_i <= c$

mi>=1 and integer i, 1<=i<=n

A dynamic programming solution may be obtained in a manner similar to that used for the knapsack problem. Since, we may assume each c_i; > 0, each m_i must be in the range $1<=m_i<=u_i$ where

$$u_i = \lfloor (c + c_i - \sum_{1 \text{ to } n} c_j) / c_i \rfloor$$

The upper bound u_i follows from the observation that $m_j>=1$. An optimal solution *m* 1 , *m* 2, ••• , m_n is the result of a sequence of decisions, one decision for each m_i.

Letfi(x) represent the maximum value of $\Phi(m_j)$, 1<=j<=i subject to the constraints $\sum_{1<=j<=i} c_j m_j <= x$ and 1<=mj<=uj, 1<=j<=i. Then, the value of an optimal solution is $f_n(c)$. The last decision made requires one to choose m_n from one of { l, 2, 3, ... , u_n.}. Once a value for m_n has been chosen, the remaining decisions must be such as to use the remaining funds $c - c_n m_n$ in an optimal way. The principal of optimality holds and

$$f_n(c) = \max_{1<=mn<=un} \{ \Phi_n(m_n) f_{n-1}(c - c_n m_n) \}$$

For any *fi(x)*,i>=1 this equation generalizes to

$$f_i(x) = \max_{1<=mi<=ui} \{ \Phi_i(m_i) f_{i-1}(c - c_i m_i) \}$$

S.NO	RGPV QUESTIONS	Year	Marks
Q.1			
Q.2			

Unit-III/Lecture-06

Problems based on Reliability design

Problems based on Reliability design:

Q.1 Design a three stage system with device types D1,D2,D3. The costs are Rs. 30, Rs. 15 and Rs. 20 respectively. The cost of the system is to be no more than Rs. 105. The reliability of each device type is 0.9,0.8 and 0.5 respectively.

Solution:

We will first compute u_1, u_2, u_3 using following formula.

$u_i = (C + C_i - \text{sigma } C_j)/ C_i$

For computing u_i

$u_1 = 2$(approx value)

For computing u_2

$u_2 = 3$(approx value)

For computing u_3

$u_3 = 3$

Hence (u_1, u_2, u_3)

Computing subsequences-

$S^0 = (1,0)$

Let Si consist of tuples of the form (f, x) =(r, c)

$S^0 = \{(1,0)\}$

For device D_1 for 1 D_1

$r_1=0.9, c_1=30$

$S^1_1 = \{(0.9, 30)\}$

For device D_1 for 2 D_1

$m_1=2$(2 D_1 device in parallel)

Reliability of stage 1 = $1-(1- r_1)^2$

Reliability of stage 1 = $1-(1- 0.9)^2 = 0.99$

Cost =30*2 = 60

$S^1_2 = \{(0.99, 60)\}$

$S^1 = \{(0.9, 30), (0.99, 60)\}$

$S^1 = \{(0.9, 30), (0.99, 60)\}$

For one Device D2:-

$S^2_1 = \{(0.72, 45), (0.792, 75)\}$

For two Device D2:-

$S^2_2 = \{(0.864, 60), (0.9504, 90)\}$

For three Device D2:-

$S^2_3 = \{(0.8928, 75), (0.98208, 105)\}$

$S^2 = \{(0.72, 45), (0.792, 75), (0.864, 60),(0.9504, 90), (0.8928, 75), (0.98208, 105)\}$

(0.792, 75), (0.9504, 90) is eleminated due to purging or dominance rule and (0.98208, 105) is eleminated due to access cost 105.

After this we got

$S^2 = \{(0.72, 45), (0.864, 60), (0.8928, 75)\}$

For one Device D3:-

$S^3_1 = \{(0.36, 65), (0.432, 80), (0.4464, 95)\}$

For Two Device D3:-

$S^3_2 = \{(0.54, 85), (0.648, 100)\}$

For Three Device D3:-

S^3_3 = { (0.63, 105) }

Now we are going to find S^3
S^3 = { (0.36, 65), (0.432, 80), (0.4464, 95), (0.54, 85), (0.648, 100), (0.63, 105) }
Due to purging rule after elimination we get
S^3 = { (0.36, 65), (0.432, 80), (0.54, 85), (0.648, 100),}
Now
The best design has a reliability of 0.648 and a cost of 100.
Tracing back through S^i's
We determine that $m_1 = 1$, $m_2 = 2$, $m_3 = 2$

S.NO	RGPV QUESTIONS	Year	Marks
Q.1			
Q.2			

Unit-III/Lecture-07

Floyd-Warshall algorithm

Floyd-Warshall algorithm:[RGPV/June-2013(7),2014(7)]
Floyd-Warshall algorithm is a procedure, which is used to find the shortest (longest) paths among all pairs of nodes in a graph, which does not contain any cycles of negative lenght. The main advantage of Floyd-Warshall algorithm is its simplicity.

Description
Floyd-Warshall algorithm uses a matrix of lengths D_0 as its input. If there is an edge between nodes i and j, than the matrix D_0 contains its length at the corresponding coordinates. The diagonal of the matrix contains only zeros. If there is no edge between edges i and j, than the position (i, j) contains positive infinity.

In other words, the matrix represents lengths of all paths between nodes that does not contain any intermediate node.
In each iteration of Floyd-Warshall algorithm is this matrix recalculated, so it contains lengths of paths among all pairs of nodes using gradually enlarging set of intermediate nodes. The matrix D_1, which is created by the first iteration of the procedure, contains paths among all nodes using exactly one (predefined) intermediate node. D_2 contains lengths using two predefined intermediate nodes. Finally the matrix D_n uses n intermediate nodes. This transformation can be described using the following recurrent formula:

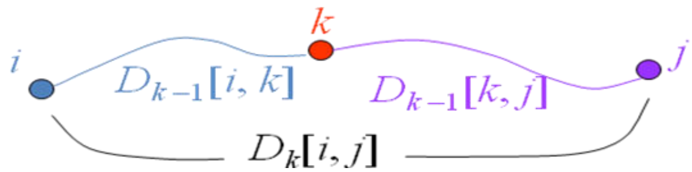
Floyd's Algorithm:
Define the notation $D_k[i, j]$, $1 \le i, j \le n$, and $0 \le k \le n$, that stands for the shortest distance (via a shortest path) from node i to node j, passing through nodes whose number (label) is $\le k$. Thus, when $k = 0$, we have

$D_0[i, j] = W[i][j]$ = the edge weight from node i to node j

This is because no nodes are numbered ≤ 0 (the nodes are numbered 1 through n). In general, when $k \ge 1$,

$$D_k[i, j] = \min(D_{k-1}[i, j], D_{k-1}[i, k] + D_{k-1}[k, j])$$

The reason for this recurrence is that when computing $D_k[i, j]$, this shortest path either doesn't go through node k, or it passes through node k exactly once. The former case yields the value $D_{k-1}[i, j]$; the latter case can be illustrated as follows:



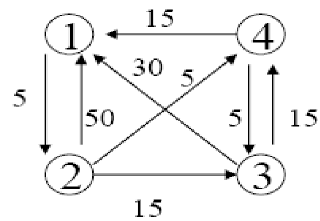
Implementation of Floyd's Algorithm:
Input: The weight matrix W[1..n][1..n] for a weighted directed graph, nodes are labeled 1 through n.
Output: The shortest distances between all pairs of the nodes, expressed in an n × n matrix.

Algorithm:
Create a matrix D and initialize it to W.

```
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])
```

Note that one single matrix D is used to store D_{k-1} and D_k, i.e., updating from D_{k-1} to D_k is done immediately. This causes no problems because in the k^{th} iteration, the value of $D_k[i, k]$ should be the same as it was in $D_{k-1}[i, k]$; similarly for the value of $D_k[k, j]$. The time complexity of the above algorithm is $O(n^3)$ because of the triple-nested loop; the space complexity is $O(n^2)$ because only one matrix is used.

Example: We demonstrate Floyd's algorithm for computing $D_k[i, j]$ for $k = 0$ through $k = 4$, for the following weighted directed graph:



Solution:

$$D_0 = W = \begin{pmatrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & \infty & 0 & 15 \\ 15 & \infty & 5 & 0 \end{pmatrix}$$

$$D_1 = \begin{pmatrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix} \qquad D_2 = \begin{pmatrix} 0 & 5 & 20 & 10 \\ 50 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

reduced from ∞ because the path (3,1,2) going thru node 1 is possible in D_1

$$D_3 = \begin{pmatrix} 0 & 5 & 20 & 10 \\ 45 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix} \qquad D_4 = \begin{pmatrix} 0 & 5 & 15 & 10 \\ 20 & 0 & 10 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$