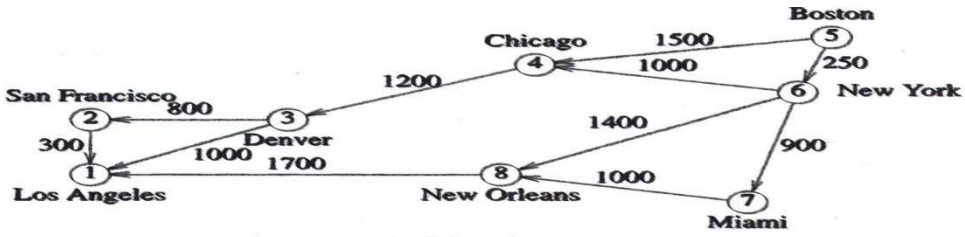
repeat
end SHORT - PATHS
Overall run time of algorithm is O ((n+|E|) log n)

Example:



(a) Digraph

	1	2	3	4	5	6	7	8
1	0							
2	300	0						
3	100	800	0					
4			1200	0				
5				1500	0	250		
6				1000		0	900	1400
7							0	1000
8	1700							0

(b) Length-adjacency matrix

Iteration	S	Vertex selected	Distance LA [1]	SF [2]	DEN [3]	CHI [4]	BOST [5]	NY [6]	MIA [7]	NO [8]
Initial	--	----	+∞	+∞	+∞	1500	0	250	+∞	+∞
1	{5}	6	+∞	+∞	+∞	1250	0	250	1150	1650
2	{5,6}	7	+∞	+∞	+∞	1250	0	250	1150	1650
3	{5,6,7}	4	+∞	+∞	2450	1250	0	250	1150	1650
4	{5,6,7,4}	8	3350	+∞	2450	1250	0	250	1150	1650
5	{5,6,7,4,8}	3	3350	3250	2450	1250	0	250	1150	1650
6	{5,6,7,4,8,3}	2	3350	3250	2450	1250	0	250	1150	1650
	{5,6,7,4,8,3,2}									

****

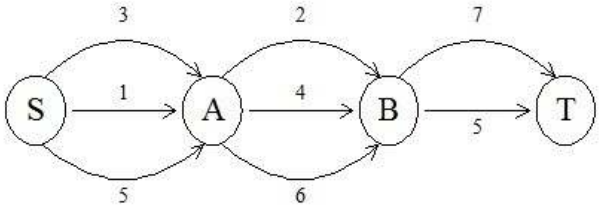
Chapter-4
Dynamic programming

4.1 The General Method

Dynamic Programming: is an algorithm design method that can be used when the solution to a problem may be viewed as the result of a sequence of decisions.

The shortest path

To find a shortest path in a multi-stage graph



Apply the greedy method

the shortest path from S to T
1 + 2 + 5 = 8

4.2 Principle of optimality

Suppose that in solving a problem, we have to make a sequence of decisions D1, D2… Dn. If this sequence is optimal, then the last k decisions, 1 k n must be optimal.

Ex: The shortest path problem

If i1, i2… j is a shortest path from i to j, then i1, i2… j must be a shortest path from i1to j
If a problem can be described by a multistage graph, then it can be solved by dynamic programming.

4.2.1 Forward approach and backward approach

Note that if the recurrence relations are formulated using the forward approach then the relations are solved backwards. i.e., beginning with the last decision

On the other hand if the relations are formulated using the backward approach, they are solved forwards.

To solve a problem by using dynamic programming

- Find out the recurrence relations.
- Represent the problem by a multistage graph.

Backward chaining vs. forward chaining

Recursion is sometimes called "backward chaining": start with the goal you want choosing your sub goals on an as-needed basis.

- Reason backwards from goal to facts (start with goal and look for support for it)
  Another option is "forward chaining": compute each value as soon as you can, in hope that you'll reach the goal.
- Reason forward from facts to goal (start with what you know and look for things you can prove)

Using forward approach to find cost of the path:

Cost (i, j) = min {c(j, *l*) + cost(i+1, *l*)}

L ϵ **Vi+1**
<j, *l*> ϵ **E**

```
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 ⟨i, j⟩. 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 ⟨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 to k − 1 do p[j] := d[p[j − 1]];
}
```

Algorithm 4.1 Multistage graph pseudo code corresponding to the forward approach Using backward approach:

Let bp (i, j) be a minimum cost path from vertex s to vertex j in Vi Let bcost (i,j) be cost of bp(i, j). The backward apporach to find minimum cost is:

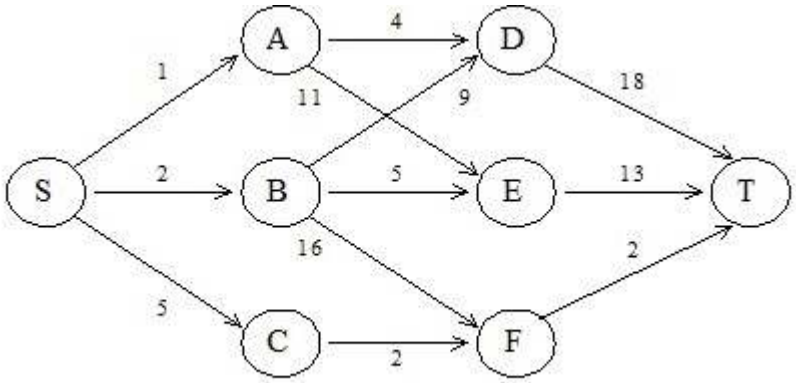
bcost (i, j) = min {bcost (i-1,*l*) +c(*l* ,j)}
l ϵ **Vi+1**
<j,*l*> ϵ **E**

Since bcost (2,j) = c(1,j) if <1,j> ϵ E and bcost (2,j) = ∞
if <i,j> ϵ E, bcost (i,j) can be computed using above formula.

```
Algorithm Bgraph(G,k,n,p)
{
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 mimimum;
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]];
}
```

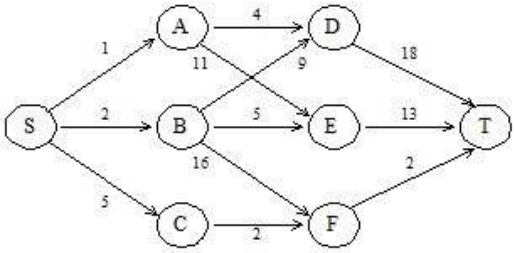
Algorithm: 4.1.1 Multi-stage graph pseudo code for corresponding backward approach.

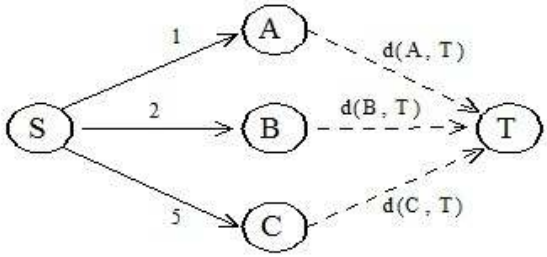
The shortest path in multistage graphs:



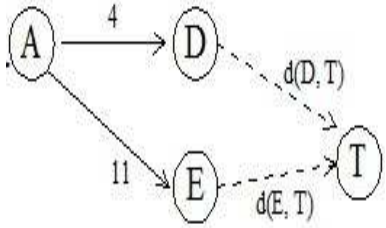
- The greedy method cannot be applied to this case: (S, A, D, T) 1+4+18 = 23.
- The real shortest path is:
  (S, C, F, T) 5+2+2 = 9.

Dynamic programming approach (forward approach)



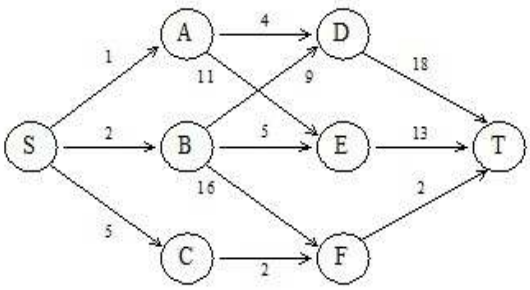


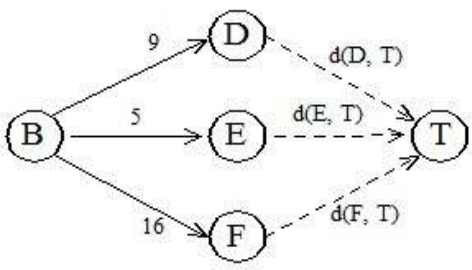
d(S, T) = min{1+d(A, T), 2+d(B, T), 5+d(C, T)}



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

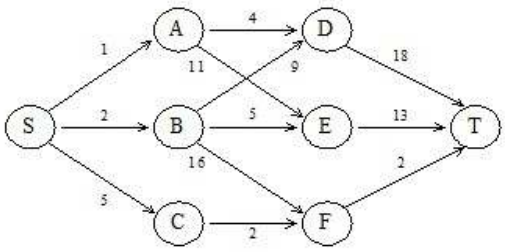
d(B, T) = min{9+d(D, T), 5+d(E, T), 16+d(F, T)}
= min{9+18, 5+13, 16+2} = 18.





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.
The above way of reasoning is called backward reasoning.
Backward approach (forward reasoning):
d(S, A) = 1
d(S, B) = 2
d(S, C) = 5
d(S,D) = min{d(S,A)+d(A,D),d(S,B)+d(B,D)}
= min{ 1+4, 2+9 } = 5
d(S,E) = min{d(S,A)+d(A,E), d(S,B)+d(B,E)}
= min{ 1+11, 2+5 } = 7
d(S,F) = min{d(S,B)+d(B,F), d(S,C)+d(C,F)}
= min{ 2+16, 5+2 } = 7
d(S,T) = min{d(S, D)+d(D, T), d(S,E)+d(E,T), d(S, F)+d(F, T)}
= min{ 5+18, 7+13, 7+2 }
= 9

4.3 Multistage Graphs



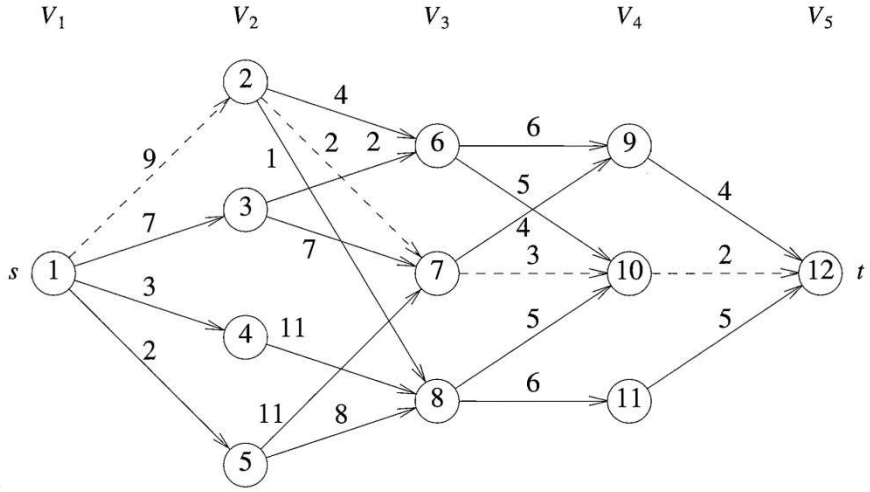
Multistage graph problem is to determine shortest path from source to destination. A multistage graph G=(V,E) is a directed graph in which the vertices are partitioned into k>=2 disjoint sets Vi, 1<=i<=k.

The vertex s is source and t is the sink. Let c(i,j) be the cost of edge <i, j>. The cost of a path from s to t is the sum of costs of the edges on the path. The multistage graph problem is to find a minimum-cost path from s to t.

A dynamic programming formulation for a 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 ith decision involve determining which vertex in Vi+1, 1<=i<=k-2, is on the path. It is easy to see that principal of optimality holds.
Let p(i,j) be a minimum-cost path from vertex j in Vi to vertex t. Let cost(i,j) be the cost of this path.

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



$$
\begin{aligned}
cost(3,6) &= \min\ \{6 + cost(4,9), 5 + cost(4,10)\} \\
&= 7 \\
cost(3,7) &= \min\ \{4 + cost(4,9), 3 + cost(4,10)\} \\
&= 5 \\
cost(3,8) &= 7 \\
cost(2,2) &= \min\ \{4 + cost(3,6), 2 + cost(3,7), 1 + cost(3,8)\} \\
&= 7 \\
cost(2,3) &= 9 \\
cost(2,4) &= 18 \\
cost(2,5) &= 15 \\
cost(1,1) &= \min\ \{9 + cost(2,2), 7 + cost(2,3), 3 + cost(2,4), \\
&\qquad 2 + cost(2,5)\} \\
&= 16
\end{aligned}
$$

The time for the **for** loop of line 7 is $\Theta(|V| + |E|)$, and the time for the **for** loop of line 16 is $\Theta(k)$, Hence, the total time is $\Theta(|V| + |E|)$.
The backward trace from vertex 1 to *n* also works.
The algorithm also works for the edges crossing more than 1 stage.

4.4 All-pairs Shortest Paths

Let G=(V,E) be a directed graph with n vertices. The cost I,j=0 if i=j , cost I,j is ∞ if i≠j, <i,j> not blelongs E

The cost i,j>0 iif i≠j <i,j> not belongs E

All pairs shortest path problem is to determine the matrix 'A' such that A(i,j) is the length of the shortest path from itoj. The matrix 'A' can be obtained by solving 'n' single source problems by using shortest path algorithm.

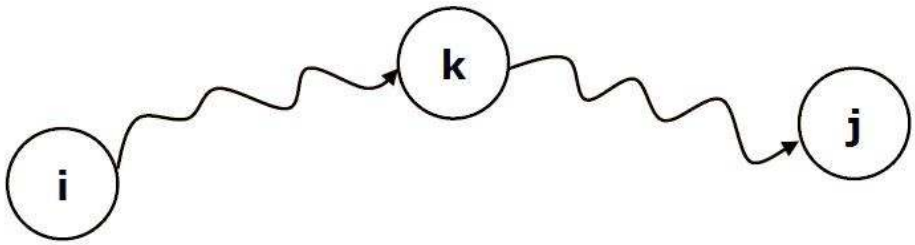
Idea

Label the vertices with integers 1..n

Restrict the shortest paths from i to j to consist of vertices 1..k only (except i and j)

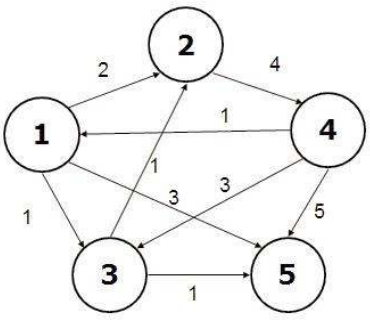
Iteratively relax k from 1 to n.

Find shortest distance from i to j using vertices 1...k

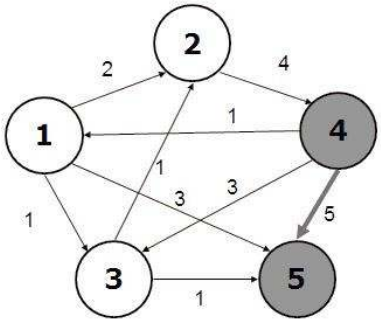


Example

i=4, j=5,k=0

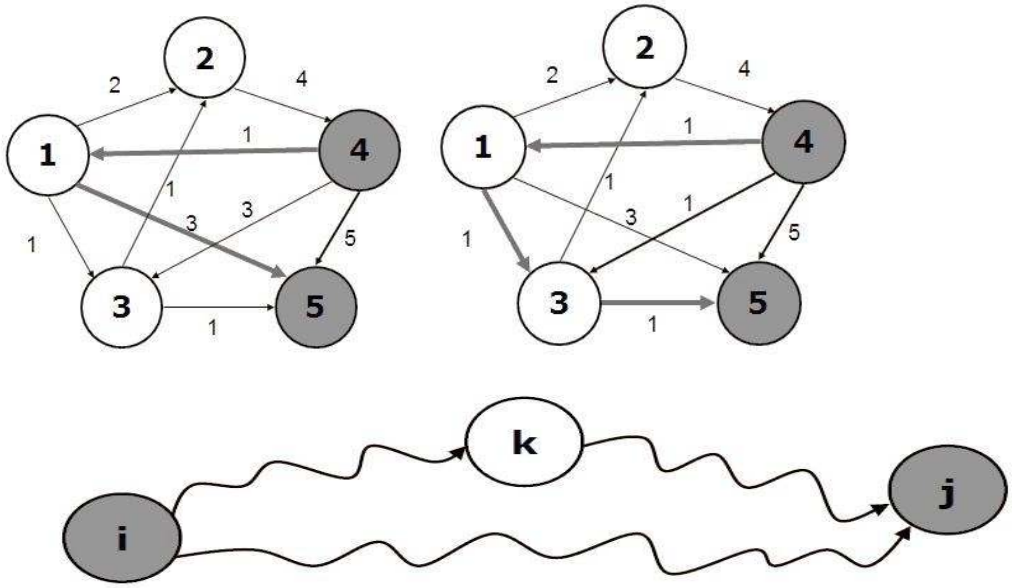


i=4, j=5,k=1



i=4, j=5,k=2

i=4, j=5,k=3



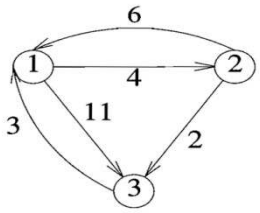
$D^k_{i,j}$: Shortest distance from i to j involving {1..k} only

$$D^k_{i,j} = \begin{cases} \min(D^{k-1}_{i,j}, D^{k-1}_{i,k} + D^{k-1}_{k,j}) & \text{for } k > 0 \\ w_{i,j} & \text{for } k = 0 \end{cases}$$

4.4.1 Shortest Path: Optimal substructure

Let G be a graph, W_{ij} be the length of edge (i, j), where $1<=i, j<= n$, and $d^{(k)}ij$ be the length of the shortest path between nodes I and j, for $1<= i, j, k<=n$, without passing through any nodes numbered greater than k.

Recurrence:



(a) Example digraph

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

At d=1

$d^1_{(1,1)} = \min\{d^0(1,1), d^0(1,1,)+ d^0(1,1,)\} = 0$

$d^1_{(1,2)} = \min\{d^0(1,2), d^0(1,1)+d^0(1,2)\} = \min\{4, 0+4\}=4$
$d^1_{(1,3)} = \min\{d^0(1,3), d^0(1,1)+d^0(1,3)\} = \min\{11,0+11\} = 11$
$d^1_{(2,1)} = \min\{d^0(2,1), d^0(2,1)+ d^0(1,1)\}$
$d^1_{(2,2)} = \min\{d^0(2,2), d^0(2,1)+d^0(1,2)\} = 0$
$d^1_{(2,3)} = \min\{d^0(2,3), d^0(2,1)+d^0(1,3)\}= \min\{2,6+11\}=2$
$d^1_{(3,1)} = \min\{d^0(3,1), d^0(3,1) + d^0(1,1)\} = \min\{3, 3+0\} = 3$
$d^1_{(3,2)} = \min\{d^0(3,2), d^0(3,1)+d^0(1,2)\}= \min\{\infty, 3+4\}=7$
$d^1_{(3,3)} = 0$

At d=2

$d^2_{(1,1)} = \min\{d^1(1,1), d^1(1,2)+ d^1(2,1,)\} = \min\{0, \infty\} = 0$
$d^2_{(1,2)} = \min\{d^1(1,2), d^1(1,2)+d^1(2,2)\} =4$
$d^2_{(1,3)} = \min\{d^1(1,3), d^1(1,2)+d^1(2,3)\} = 6$
$d^2_{(2,1)} = \min\{d^1(2,1), d^1(2,2)+ d^1(2,1)\} =6$
$d^2_{(2,2)} = \min\{d^1(2,2), d^1(2,1)+d^1(2,2)\} = \min\{0, 0+0\} = 0$
$d^2_{(2,3)} = \min\{d^1(2,3), d^1(2,1)+d^1(2,3)\}= 2$
$d^2_{(3,1)} = \min\{d^1(3,1), d^1(3,2) + d^1(2,1)\} = \min\{3, 7+6\} = 3$
$d^2_{(3,2)} = \min\{7, 7+0\}=7$
$d^2_{(3,3)} = 0$

At d=3

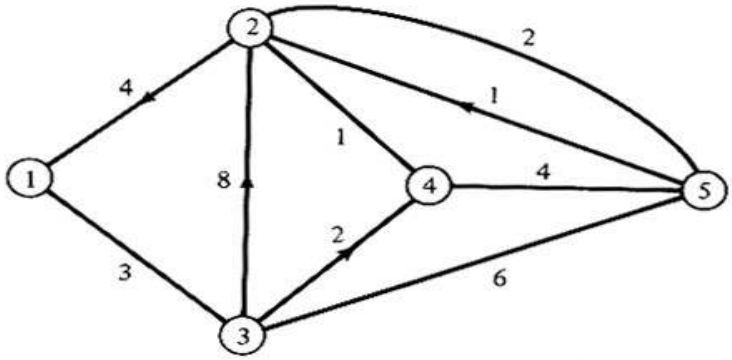
$d^3_{(1,1)} = \min\{0, somevalue\} = 0$
$d^3_{(1,2)} = \min\{4,6+0\}=4$
$d^3_{(1,3)} = \min\{6,6+0)\} = 6$
$d^3_{(2,1)} = \min\{6, 2+3\}=5$
$d^3_{(2,2)} = 0$
$d^3_{(2,3)} = \min\{2,2\}=2$
$d^3_{(3,1)} = \min\{3, 3+0\} = 3$
$d^3_{(3,2)} = \min\{7, 7+0\}=7$
$d^3_{(3,3)} = 0$

```
0   Algorithm AllPaths(cost, A, n)
1   // cost[1 : n, 1 : n] is the cost adjacency matrix of a graph with
2   // n vertices; A[i, j] is the cost of a shortest path from vertex
3   // i to vertex j. cost[i, i] = 0.0, for 1 ≤ i ≤ n.
4   {
5       for i := 1 to n do
6           for j := 1 to n do
7               A[i, j] := cost[i, j]; // Copy cost into A.
8       for k := 1 to n do
9           for i := 1 to n do
10              for j := 1 to n do
11                  A[i, j] := min(A[i, j], A[i, k] + A[k, j]);
12  }
```

Algorithm 4.2 All-Pairs Shortest Paths algorithm

- Find the distance between every pair of vertices in a weighted directed graph G.
- We can make n calls to Dijkstra's algorithm (if no negative edges), which takes O(nmlog n) time.
- Likewise, n calls to Bellman-Ford would take O (n2m) time.
- We can achieve O (n3) time using dynamic programming (similar to the Floyd-Warshall algorithm).

Example for all pairs shortest path:



Initialization:

$D^{(0)}$ =

	1	2	3	4	5
1	0	∞	3	∞	∞
2	4	0	∞	1	2
3	3	8	0	2	6
4	∞	1	∞	0	4
5	∞	1	6	4	0

$P^{(0)}$ =

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

Through Node 1:

$D^{(1)}$ =

	1*	2	3	4	5
*1	0	∞	3	∞	∞
2	4	0	7^+	1	2
3	3	8	0	2	6
4	∞	1	∞	0	4
5	∞	1	6	4	0

$P^{(1)}$ =

	1*	2	3	4	5
*1	—	1	1	1	1
2	2	—	1^+	2	2
3	3	3	—	3	3
4	4	4	4	—	4
5	5	5	5	5	—

Through Node 2:

$D^{(2)}$ =

	1	2*	3	4	5
1	0	∞	3	∞	∞
*2	4	0	7	1	2
3	3	8	0	2	6
4	5+	1	8+	0	3+
5	5+	1	6	2+	0

$P^{(2)}$ =

	1	2*	3	4	5
1	—	1	1	1	1
*2	2	—	1	2	2
3	3	3	—	3	3
4	2+	4	1+	—	2+
5	2+	5	5	2+	—

Through Node 3:

$D^{(3)}$ =

	1	2	3*	4	5
1	1	11+	3	5+	9+
2	4	0	7	1	2
*3	3	8	0	2	6
4	5	1	8	0	3
5	5	1	6	2	0

$P^{(3)}$ =

	1	2	3*	4	5
1	—	3+	1	3+	3+
2	2	—	1	2	2
*3	3	3	—	3	3
4	2	4	1	—	2
5	2	5	5	2	—

Through Node 4:

$D^{(4)}$ =

	1	2	3	4*	5
1	0	6+	3	5	8+
2	4	0	7	1	2
3	3	3+	0	2	5+
*4	5	1	8	0	3
5	5	1	6	2	0

$P^{(4)}$ =

	1	2	3	4*	5
1	—	4+	1	3	2+
2	2	—	1	2	2
3	3	4+	—	3	2+
*4	2	4	1	—	2
5	2	5	5	3	—

Through Node 5:

$D^{(5)}$ =

	1	2	3	4	5*
1	0	6	3	5	8
2	4	0	7	1	2
3	3	3	0	2	5
4	5	1	8	0	3
*5	5	1	6	2	0

$P^{(5)}$ =

	1	2	3	4	5*
1	—	4	1	3	2
2	2	—	1	2	2
3	3	4	—	3	2
4	2	4	1	—	2
*5	2	5	5	2	—

Note that on the last pass no improvements could be found for $D^{(5)}$ over $D^{(4)}$. The final matrices $D^{(5)}$ and $P^{(5)}$ indicate, for instance, that the shortest path from node 1 to node 5 has length d(1,5) = 8 units and that this shortest path is the path {1, 3, 4, 2, 5}.

To identify that shortest path, we examined row 1 of the $P^{(5)}$ matrix. Entry p_5 says that the predecessor node to 5 in the path from 1 to 5 is node 2; then, entry p_5 (1, 2) says that the predecessor node to 2 in the path from 1 to 2 is node 4; similarly, we backtrack the rest of the path by examining p_5(1, 4) (= 3) and p_5(1, 3) = 1. In general, backtracking stops when the predecessor node is the same as the initial node of the required path.

For another illustration, the shortest path from node 4 to node 3 is d (4, 3) = 8 units long and the path is {4, 2, 1, 3}. The predecessor entries that must be read are, in order, p_5 (4, 3) = 1, p_5 (4, 1) = 2, and finally p_5(4, 2) = 4--at which point we have "returned" to the initial node.

4.5 Single-Source Shortest Paths

4.5.1 General Weights

Let $dist^k[u]$ be the length of a shortest path from the source vertex v to vertex u containing at most k edges.

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

```
1   Algorithm BellmanFord(v, cost, dist, n)
2   // Single-source/all-destinations shortest
3   // paths with negative edge costs
4   {
5       for i := 1 to n do // Initialize dist.
6           dist[i] := cost[v, i];
7       for k := 2 to n - 1 do
8           for each u such that u ≠ v and u has
9                   at least one incoming edge do
10              for each ⟨i, u⟩ in the graph do
11                  if dist[u] > dist[i] + cost[i, u] then
12                      dist[u] := dist[i] + cost[i, u];
13  }
```

Algorithm 4.3 Bellman and ford algorithm to compute shortest paths

Bellman and ford algorithm: Works even with negative-weight edges

It must assume directed edges (for otherwise we would have negative-weight cycles)

Iteration i finds all shortest paths that use i edges.

Running time: O(nm).
It Can be extended to detect a negative-weight cycle if it exists.

4.6 Optimal Binary Search Trees

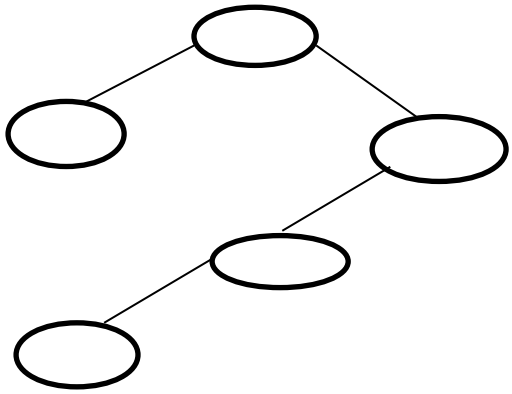
Definition: Binary search tree (BST) A binary search tree is a binary tree; either it is empty or each node contains an identifier and

1. All identifiers in the left sub tree of T are less than the identifiers in the root node T.
2. All the identifiers the right sub tree is greater than the identifier in the root node T.
3. The right and left sub tree are also BSTs.

Algorithm for searching an identifier in the tree 'T'

```
Procedure SEARCH (T X I)
// Search T for X, each node had fields LCHILD, IDENT, RCHILD//
// Return address I pointing to the identifier X// //Initially T is pointing to tree.
//ident(i)=X or i=0
//I ← T
While I ≠ 0 do
  case : X < Ident(i) : I ←LCHILD(i)
       : X = IDENT(i) : RETURN i
         : X > IDENT(i) : I ← RCHILD(i)
  end case
repeat
end SEARCH
```

Optimal Binary Search trees – Example



If each identifier is searched with equal probability the average number of comparisons for the above tree is 1+2+2+3+4/5 = 12/5.

- Let us assume that the given set of identifiers are $\{a_1, a_2...a_n\}$ with $a_1<a_2<.....<a_n$.
- Let P_i be the probability with which we are searching for a_i.
- Let Q_i be the probability that identifier x being searched for is such that $a_i<x<a_{i+1}$ $0\leq i\leq n$, and $a_0=-\infty$ and $a_{n+1}=+\infty$.
- Then $\sum Q_i$ is the probability of an unsuccessful search.

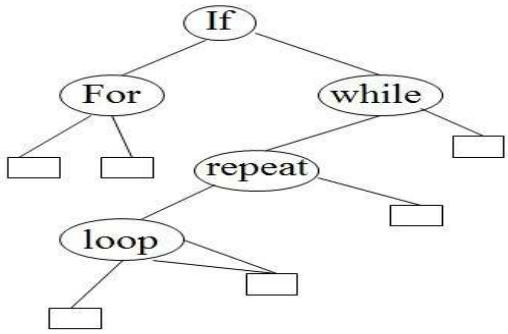
$0\leq i\leq n$

$\sum P(i) + \sum Q(i) = 1$. Given the data,

$1\leq i\leq n \quad 0\leq i\leq n$

let us construct one optimal binary search tree for $(a_1.....a_n)$.

- In place of empty sub tree, we add external nodes denoted with squares.
- Internet nodes are denoted as circles.



4.7 Construction of optimal binary search trees

i) A BST with n identifiers will have n internal nodes and n+ 1 external node.
ii)Successful search terminates at internal nodes unsuccessful search terminates at external nodes.
iii)If a successful search terminates at an internal node at level L, then L iterations of the loop in the algorithm are needed.
iv)Hence the expected cost contribution from the internal nodes for a_i is P (i) * level (a_i).
v)Unsuccessful search terminates at external nodes i.e. at i = 0.
vi)The identifiers not in the binary search tree may be partitioned into n+1 equivalent classes

$E_i \quad 0\leq i\leq n$.

E_o contains all X such that $X\leq a_i$

E_i contains all X such that $a<X<=a_{i+1}$ $1\le i\le n$

E_n contains all X such that $X > a_n$

For identifiers in the same class E_i, the search terminates at the same external node. If the failure node for E_i is at level L, then only L-1 iterations of the while loop are made

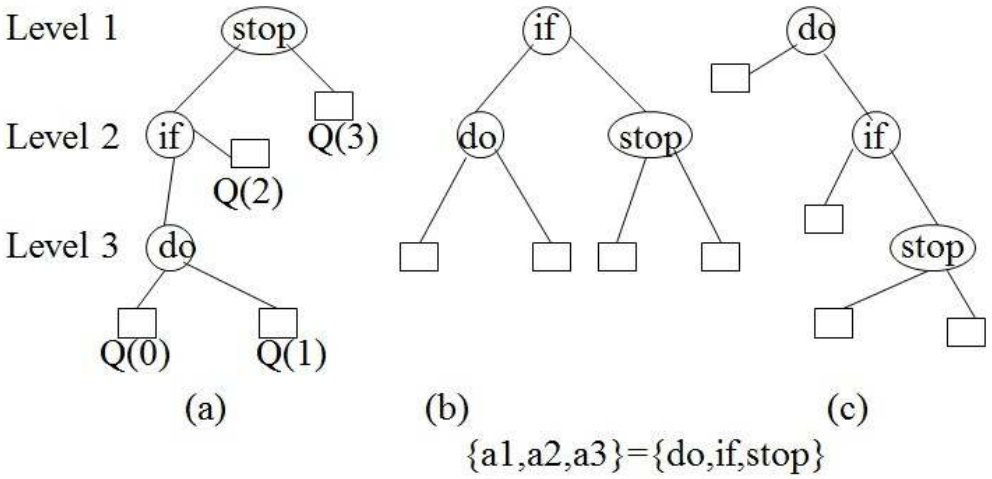
$\therefore$ The cost contribution of the failure node for E_i is Q (i) * level (E_i) -1)

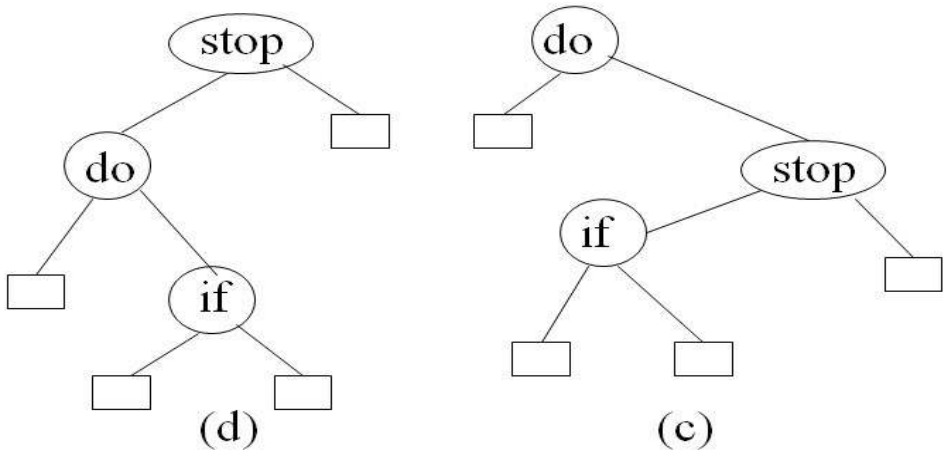
Thus the expected cost of a binary search tree is:

$$\sum_{1\le i\le n} P(i) * \text{level}(a_i) + \sum_{0\le i\le n} Q(i) * \text{level}(E_i) - 1) \quad \ldots\ldots(2)$$

An optimal binary search tree for $\{a_1\ldots\ldots, a_n\}$ is a BST for which (2) is minimum.

Example: Let $\{a_1, a_2, a_3\}$ ={do, if, stop}





With equal probability P (i) = Q(i) = 1/7.

Let us find an OBST out of these.

Cost(tree a) = $\sum_{1\le i\le n} P(i)*\text{level a}(i) + \sum_{0\le i\le n} Q(i)*\text{level (Ei) -1}$

(2-1) (3-1) (4-1) (4-1)

=1/7[1+2+3 + 1 + 2 + 3 + 3] = 15/7

Cost (tree b) =17[1+2+2+2+2+2+2] =13/7

Cost (tree c) =cost (tree d) =cost (tree e) =15/7

∴ tree b is optimal.

If P(1) =0.5 ,P(2) =0.1, P(3) =0.005 , Q(0) =.15 , Q(1) =.1, Q(2) =.05 and Q(3) =.05 find the OBST.

Cost (tree a) = .5 x 3 +.1 x 2 +.05 x 3 +.15x3 +.1x3 +.05x2 +.05x1 = 2.65

Cost (tree b) =1.9 , Cost (tree c) =1.5 ,Cost (tree d) =2.05 , Cost (tree e) =1.6.

Hence tree C is optimal.

To obtain a OBST using Dynamic programming we need to take a sequence of decisions regard. The construction of tree.

First decision is which of a_i is being as root.

Let us choose a_k as the root. Then the internal nodes for $a_1,\ldots\ldots,a_{k-1}$ and the external nodes for classes $E_o,E_1,\ldots\ldots,E_{k-1}$ will lie in the left sub tree L of the root. The remaining nodes will be in the right sub tree R.

Define

$$\text{Cost (L)} = \sum_{1\le i\le k} P(i) * \text{level}(a_i) + \sum_{0\le i\le k} Q(i)*(\text{level}(E_i)-1)$$

$$\text{Cost(R)} = \sum_{k\le i\le n} P(i)*\text{level}(a_i) + \sum_{k\le i\le n} Q(i)*(\text{level}(E_i)-1)$$

Tij be the tree with nodes $a_{i+1},\ldots..,a_j$ and nodes corresponding to $E_i,E_{i+1},\ldots..,E_j$.

Let W (i, j) represents the weight of tree T_{ij}.

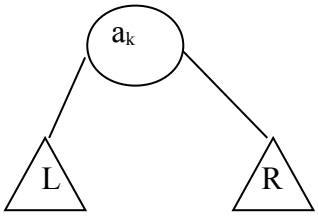
$$W(i, j)=P(i+1) +\ldots+P(j)+Q(i)+Q(i+1)\ldots Q(j)=Q(i) +\sum_{l=i+1}^{j} [Q(l)+P(l)]$$

The expected cost of the search tree in (a) is (let us call it T) is

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

W (0, k-1) is the sum of probabilities corresponding to nodes and nodes belonging to equivalent classes to the left of a_k.

W (k, n) is the sum of the probabilities corresponding to those on the right of a_k.



(a) OBST with root a_k

4.8 0-1 Knapsack

If we are given 'n' objects and a knapsack or a bag, in which the object ' i' has weight 'w_i' is to be placed, the knapsack has capacity 'N' then the profit that can be earned is $p_i x_i$. The objective is to obtain filling of knapsack with maximum profits is to

$$\text{maximize} \sum_{1 \le i \le n} p_i x_i$$

$$\text{subject to} \sum_{1 \le i \le n} w_i x_i \le m$$

$$\text{and } 0 \le x_i \le 1, \quad 1 \le i \le n$$

n=no of objects i=1,2,….n.; m=capacity of the bag ;
w_i =weight of object I; P_i =profit of the object i.

In solving 0/1 knapsack problem two rules are defined to get the solution.
Rule1: When the weight of object(s) exceeds bag capacity than discard that pair(s).
Rule2: When (p_i,w_i) and (p_j,w_j) where pi $\le p_j$and $w_i \ge$wjthan (p_i,w_i) pair will be discarded. This rule is called purging or dominance rule.Applying dynamic programming method to calculate 0/1 knapsack problem the formula equation is: $S_1^i = \{(P,W)|(P-p_i , W-w_i) \epsilon S^i \}$

Example-1
N=3 ; m=6 (p1,p2,p3)=(1,2,5)
(w1,w2,w3)=(2,3,4) Rule 1:
Initially $S^0 = \{0,0\}$

$S^0{}_1 = \{1,2\}$
$S^1 = S^0 \cup S^0{}_1 = \{(0,0),(1,2);$
$S^1{}_1 = \{(2,3),(3,5)\}$
$S^2 = S^1 \cup S^1{}_1 = \{(0,0),(1,2),(2,3),(3,5)\}$
$S^2{}_1 = \{(5,4),(6,6),(7,7),(8,9)\}$
$S^3 = S^2 \cup S^2{}_1 = \{(0,0),(1,2),(2,3),(3,5),(5,4),(6,6),(7,7),(8,9)\}$
N=3, $2^3 = 8$ pairs

Applying rule to the above pairs, where weight exceeds knapsack capacity discards the pair. In the above (7, 7), (8,9) pairs are discarded.

Rule 2(purging or dominance): Applying rule to the remaining pairs after discarded pairs i.e on 6 pairs

Pairs 3≤5 and 5≥4 pairs in above shown so that pair (3,5) pair discarded.
So the solution pair(s) is (6,6) ;
Solution vector: (p1,p2,p3)=(1,2,5)=>(p1,p3)=(1,5)
(w1,w2,w3)=(2,3,4)=>(w1,w3)=(2,4)
The solution vector is (1,0,1)

```
Alogirthm DKP(p,w,n,m)
{
S^0 :={(0,0)};
for i:= 1 to n-1 do
{
S^i-1 := {(P,W| (P-p_i , W-w_i ) ϵ S^i-1 and W ≤ m};
S^i :=MergePurge(S S^i-1 S^i-1 _1 );
}
(PX,WX):=lastpair in S^n-1 ;
(PX,WX):=(P + p_n , W + w_n ) where W is the largest W in any pair in S^n-1 such
that W + w_n ≤m;
//Trace back for x_n, x_n-1,.....x_1

If (PX>PY) then X_n:= 0;
else X_n :=1;
TraceBackFor(x_n-1,.....x_1 );
}
```

Fig. Informal Knapsack algorithm

4.9 Traveling Salesperson Problem

A salesperson would like to travel from one city to the other (n–1) cities just once then back to the original city, what is the minimum distance for this travel?

The brute-and-force method is trying all possible (n–1)! Permutations of (n–1) cities and picking the path with the minimum distance.

There are a lot of redundant computations for the brute-and-force method such as the permutations of 6 cities are 1234561, …, 1243561, …., 1324561, …, 1342561, …, 1423561, …, 1432561, …

The function g(1,V-{1}) is the length of an optimal salesperson tour. From the principal of optimality it follows that g (1, V-[1}) = $\min_{2\le k\le n}$ {c1k + g{k,V-{1,k})}
Generalizing above one we obtain (for I not belong S) g (i, S) = $\min_{jE\ s}${c_{iJ}+ g(j, S-{j})}
The method computes from vertex 1 backward to the other vertices then return to vertex 1.

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.

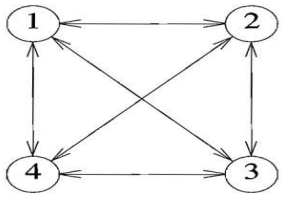
The function g(1,V-{1}) is the length of an optimal salesperson tour. From the principal of optimality it follows that

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

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

Eqn. 1 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 eqn. 2 clearly, g (i, ø) = c_{i1}, 1≤i ≤n. Hence we use g (i, S) for all S size 1.then g (i, S) for S=2 and so on.

Example



(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 (5.21), we obtain

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

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

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

Finally, from (5.20) we obtain

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

4.10 Flow shop scheduling

Let n be the no. of jobs, each may be requiring m task.

Let T_{1i}, T_{2i}… T_{mi} where i=1 to n to be performed where T_{mi} is the I^{th} job of m^{th} task. The task 'T_{ji} to be processed on the processor 'T_j' where j=1,2,……m. The time required to complete the task T_{ji} is t_{ji} .

A schedule for 'n' jobs is an assignment of tasks to the time intervals on the processors.

Constraints: No two processors may have more than one task assign to it in anytime interval.

Objective: The objective of flow shop scheduling is to find the optimal finishing time (OFT) of the given schedule 'S'.

Formula: $F(s) = \max_{1 \le i \le n}\{f_i(S)\}$

(Optimal Finishing Time OFT schedule 'S')
Mean flow time for the schedule 'S' is

$$\text{MFT} = 1/n \sum_{1 \le i \le n} (f_i(S))$$

There are 2 possible scheduling:

1) Non-preemptive schedule:-it is in which the processing of a task on any processor is not terminate until the task is completed.
2) Preemptive scheduling:-It is in which the processing of a task on any processor is terminated before the task is completed.

Problem:

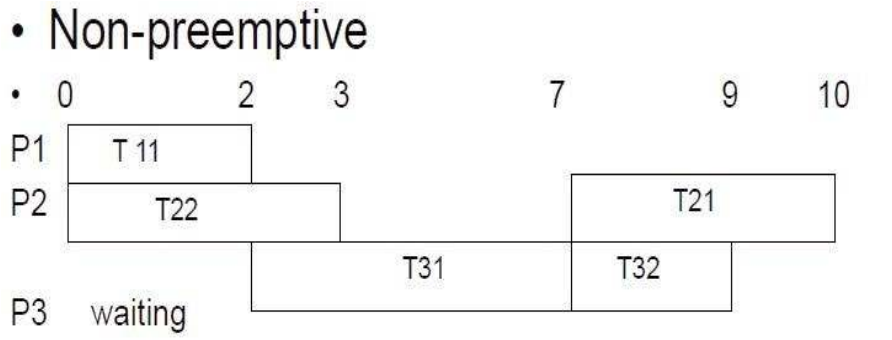
j =

2	0	p1
3	3	p2
5	2	p3

j=schedule t11

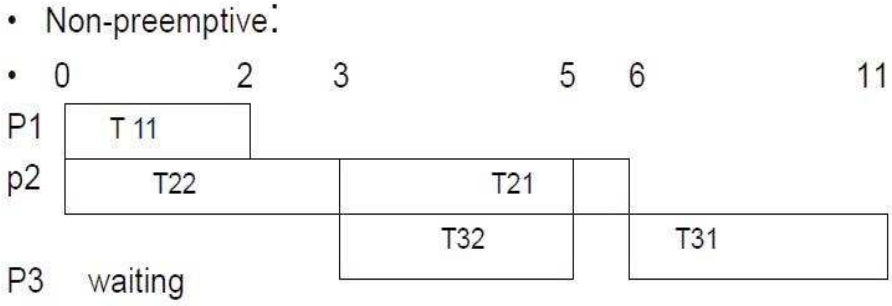
P1, p2, p3= processors or tasks of jobs

T11 –first job of the first task



Optimal Finishing Time=10

Mean Flow Time =1/2(9+10)=9.5



OFS =11

MFT = ½(min+max)=1/2(6+11)=8.5

****