UNIT III:

Greedy method: General method, applications - Job sequencing with deadlines, 0/1 knapsack problem, Minimum cost spanning trees, Single source shortest path problem. **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.

Greedy Method:

The greedy method is perhaps (maybe or possible) the most straight forward design technique, used to determine a feasible solution that may or may not be optimal.

Feasible solution:- Most problems have n inputs and its solution contains a subset of inputs that satisfies a given constraint(condition). Any subset that satisfies the constraint is called feasible solution.

Optimal solution: To find a feasible solution that either maximizes or minimizes a given objective function. A feasible solution that does this is called optimal solution.

The greedy method suggests that an algorithm works in stages, considering one input at a time. At each stage, a decision is made regarding whether a particular input is in an optimal solution.

Greedy algorithms neither postpone nor revise the decisions (ie., no back tracking). **Example**: Kruskal's minimal spanning tree. Select an edge from a sorted list, check, decide, and never visit it again.

Application of Greedy Method:

- Job sequencing with deadline
- 0/1 knapsack problem
- Minimum cost spanning trees
- Single source shortest path problem.

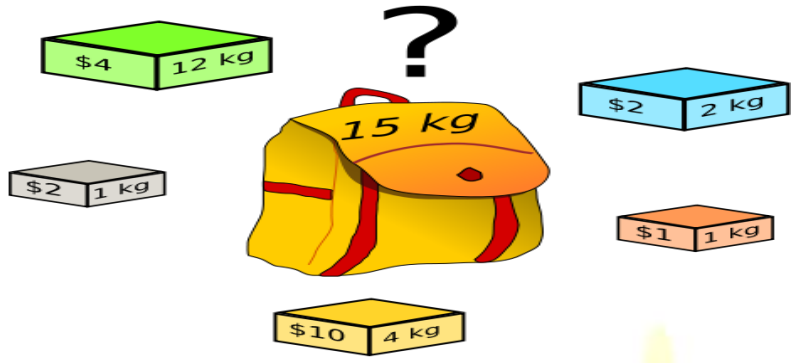
Algorithm for Greedy method

Algorithm Greedy(a,n)
//a[1:n] contains the n inputs.
{
Solution :=0;
For i=1 to n do
{
X:=select(a);
If Feasible(solution, x) then
Solution :=Union(solution,x);
}
Return solution;
}

Selection → Function, that selects an input from a[] and removes it. The selected input's value is assigned to x.
Feasible → Boolean-valued function that determines whether x can be included into the solution vector.
Union → function that combines x with solution and updates the objective function.

Knapsack problem

The **knapsack problem** or **rucksack (bag) problem** is a problem in combinatorial optimization: Given a set of items, each with a mass and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible



There are two versions of the problems

1. 0/1 knapsack problem
2. Fractional Knapsack problem
    a. Bounded Knapsack problem.
    b. Unbounded Knapsack problem.

Solutions to knapsack problems

- **Brute-force approach**:-Solve the problem with a straight farward algorithm
- **Greedy Algorithm**:- Keep taking most valuable items until maximum weight is reached or taking the largest value of eac item by calculating $v_i=value_i/Size_i$
- **Dynamic Programming**:- Solve each sub problem once and store their solutions in an array.

0/1 knapsack problem:

Let there be n items, z_1 to z_n where z_i has a value v_i and weight w_i . The maximum weight that we can carry in the bag is *W*. It is common to assume that all values and weights are nonnegative. To simplify the representation, we also assume that the items are listed in increasing order of weight.

Maximize $\sum_{i=1}^{n} v_i x_i$ subject to $\sum_{i=1}^{n} w_i x_i \leqslant W, \qquad x_i \in \{0,1\}$

Maximize the sum of the values of the items in the knapsack so that the sum of the weights must be less than the knapsack's capacity.

Greedy algorithm for knapsack

Algorithm GreedyKnapsack(m,n)
// p[i:n] and [1:n] contain the profits and weights respectively
// if the n-objects ordered such that p[i]/w[i]>=p[i+1]/w[i+1], m→ size of knapsack and x[1:n]→ the solution vector
{
For i:=1 to n do x[i]:=0.0
U:=m;
For i:=1 to n do
{
if(w[i]>U) then break;
x[i]:=1.0;
U:=U-w[i];
}
If(i<=n) then x[i]:=U/w[i];
}

Ex: - Consider 3 objects whose profits and weights are defined as
(P_1, P_2, P_3) = (25, 24, 15)
$W_1, W_2, W_3)$ = (18, 15, 10)
n=3→number of objects
m=20→Bag capacity

Consider a knapsack of capacity 20. Determine the optimum strategy for placing the objects in to the knapsack. The problem can be solved by the greedy approach where in the inputs are arranged according to selection process (greedy strategy) and solve the problem in stages. The various greedy strategies for the problem could be as follows.

(x_1, x_2, x_3)	$\sum x_i w_i$	$\sum x_i p_i$
(1, 2/15, 0)	$18x1+\frac{2}{15}x15 = 20$	$25x1+\frac{2}{15}x\ 24 = 28.2$
(0, 2/3, 1)	$\frac{2}{3}x15+10x1= 20$	$\frac{2}{3}x\ 24\ +15x1 = 31$

(0, 1, ½)	1x15+ $\frac{1}{2}$ x10 = 20	1x24+ $\frac{1}{2}$ x15 = 31.5
(½, ⅓, ¼)	½ x 18+⅓ x15+ ¼ x10 = 16. 5	½ x 25+⅓ x24+ ¼ x15 = 12.5+8+3.75 = 24.25

Analysis: - If we do not consider the time considered for sorting the inputs then all of the three greedy strategies complexity will be O(n).

Job Sequence with Deadline:

There is set of n-jobs. For any job i, is a integer deadling di≥0 and profit Pi>0, the profit Pi is earned iff the job completed by its deadline.

To complete a job one had to process the job on a machine for one unit of time. Only one machine is available for processing jobs.

A feasible solution for this problem is a subset J of jobs such that each job in this subset can be completed by its deadline.

The value of a feasible solution J is the sum of the profits of the jobs in J, i.e., $\sum_{i \in j} P_i$

An optimal solution is a feasible solution with maximum value.

The problem involves identification of a subset of jobs which can be completed by its deadline. Therefore the problem suites the subset methodology and can be solved by the greedy method.

Ex: - Obtain the optimal sequence for the following jobs.

j_1 j_2 j_3 j_4

(P_1, P_2, P_3, P_4) = (100, 10, 15, 27)

(d_1, d_2, d_3, d_4) = (2, 1, 2, 1)

n =4

Feasible solution	Processing sequence	Value
j_1 j_2 (1, 2)	(2,1)	100+10=110
(1,3)	(1,3) or (3,1)	100+15=115
(1,4)	(4,1)	100+27=127
(2,3)	(2,3)	10+15=25
(3,4)	(4,3)	15+27=42
(1)	(1)	100
(2)	(2)	10
(3)	(3)	15
(4)	(4)	27

In the example solution '3' is the optimal. In this solution only jobs 1&4 are processed and the value is 127. These jobs must be processed in the order j4 followed by j1. the process of job 4 begins at time 0 and ends at time 1. And the processing of job 1 begins at time 1 and ends at time2. Therefore both the jobs are completed within their deadlines. The optimization measure for determining the next job to be selected in to the solution is according to the profit. The next job to include is that which increases ∑pi the most, subject to the constraint that the resulting "j" is the feasible solution. Therefore the greedy strategy is to consider the jobs in decreasing order of profits.

The greedy algorithm is used to obtain an optimal solution.

We must formulate an optimization measure to determine how the next job is chosen.

```
algorithm js(d, j, n)
//d→ dead line, j→subset of jobs ,n→ total number of jobs
// d[i]≥1 1 ≤ i ≤ n are the dead lines,
// the jobs are ordered such that p[1]≥p[2]≥---≥p[n]
//j[i] is the ith job in the optimal solution 1 ≤ i ≤ k, k→ subset range
{
d[0]=j[0]=0;
j[1]=1;
k=1;
for i=2 to n do{
r=k;
while((d[j[r]]>d[i]) and [d[j[r]]≠r)) do
r=r-1;
if((d[j[r]]≤d[i]) and (d[i]> r)) then
{
for q:=k to (r+1) setp-1 do j[q+1]= j[q];
j[r+1]=i;
k=k+1;
}
}
return k;
}
```

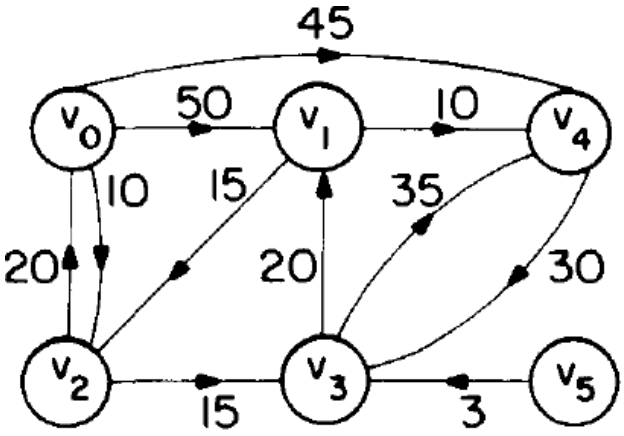
Note: The size of sub set j must be less than equal to maximum deadline in given list.

Single Source Shortest Paths:

- Graphs can be used to represent the highway structure of a state or country with vertices representing cities and edges representing sections of highway.
- The edges have assigned weights which may be either the distance between the 2 cities connected by the edge or the average time to drive along that section of highway.
- For example if A motorist wishing to drive from city A to B then we must answer the following questions
  - Is there a path from A to B
  - If there is more than one path from A to B which is the shortest path
- The length of a path is defined to be the sum of the weights of the edges on that path.

Given a directed graph G(V,E) with weight edge w(u,v). e have to find a shortest path from source vertex S$\in$v to every other vertex v1$\in$ V−S.

- To find SSSP for directed graphs G(V,E) there are two different algorithms.
  - Bellman-Ford Algorithm
  - Dijkstra's algorithm
- Bellman-Ford Algorithm:- allow –ve weight edges in input graph. This algorithm either finds a shortest path form source vertex $\mathbf{S} \in \mathbf{V}$ to other vertex $\mathbf{v} \in \mathbf{V}$ or detect a –ve weight cycles in G, hence no solution. If there is no negative weight cycles are reachable form source vertex $\mathbf{S} \in \mathbf{V}$ to every other vertex $\mathbf{v} \in \mathbf{V}$
- Dijkstra's algorithm:- allows only +ve weight edges in the input graph and finds a shortest path from source vertex $\mathbf{S} \in \mathbf{V}$ to every other vertex $\mathbf{v} \in \mathbf{V}$.



	Path	Length
1)	$v_0\, v_2$	10
2)	$v_0\, v_2\, v_3$	25
3)	$v_0\, v_2\, v_3\, v_1$	45
4)	$v_0\, v_4$	45

Graph and shortest paths from v_0 to all destinations

- Consider the above directed graph, if node 1 is the source vertex, then shortest path from 1 to 2 is 1,4,5,2. The length is 10+15+20=45.
- To formulate a greedy based algorithm to generate the shortest paths, we must conceive of a multistage solution to the problem and also of an optimization measure.
- This is possible by building the shortest paths one by one.
- As an optimization measure we can use the sum of the lengths of all paths so far generated.
- If we have already constructed 'i' shortest paths, then using this optimization measure, the next path to be constructed should be the next shortest minimum length path.
- The greedy way to generate the shortest paths from Vo to the remaining vertices is to generate these paths in non-decreasing order of path length.
- For this 1st, a shortest path of the nearest vertex is generated. Then a shortest path to the 2nd nearest vertex is generated and so on.

Algorithm for finding Shortest Path

```
Algorithm ShortestPath(v, cost, dist, n)
//dist[j], 1≤j≤n, is set to the length of the shortest path from vertex v to vertex j in graph g with n-vertices.
// dist[v] is zero
{
for i=1 to n do{
s[i]=false;
dist[i]=cost[v,i];
}
s[v]=true;
dist[v]:=0.0; // put v in s
for num=2 to n do{
// determine n-1 paths from v
choose u form among those vertices not in s such that dist[u] is minimum.
s[u]=true; // put u in s
for (each w adjacent to u with s[w]=false) do
if(dist[w]>(dist[u]+cost[u, w])) then
dist[w]=dist[u]+cost[u, w];
}
}
```

Minimum Cost Spanning Tree:

SPANNING TREE: - A Sub graph 'n' of o graph 'G' is called as a spanning tree if

(i) It includes all the vertices of 'G'

(ii) It is a tree

Minimum cost spanning tree: For a given graph 'G' there can be more than one spanning tree. If weights are assigned to the edges of 'G' then the spanning tree which has the minimum cost of edges is called as minimal spanning tree.

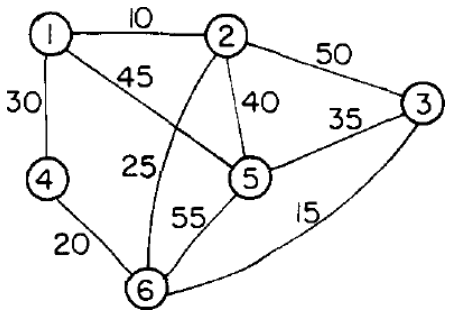
The greedy method suggests that a minimum cost spanning tree can be obtained by contacting the tree edge by edge. The next edge to be included in the tree is the edge that results in a minimum increase in the some of the costs of the edges included so far.

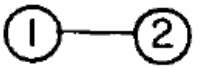
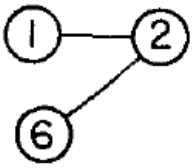
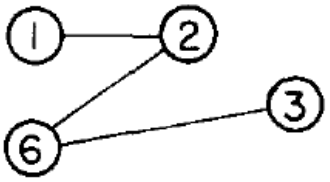
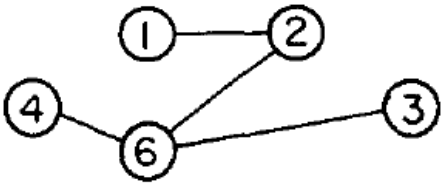
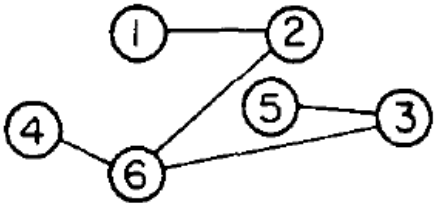
There are two basic algorithms for finding minimum-cost spanning trees, and both are greedy algorithms

→Prim's Algorithm

→Kruskal's Algorithm

Prim's Algorithm: Start with any *one node* in the spanning tree, and repeatedly add the cheapest edge, and the node it leads to, for which the node is not already in the spanning tree.



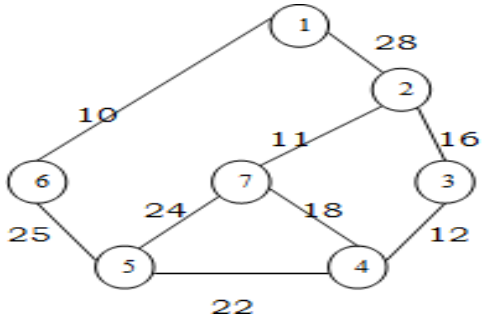
Edge	Cost	Spanning tree
(1,2)	10	
(2,6)	25	
(3,6)	15	
(6,4)	20	
(1,4)	reject	
(3,5)	35	

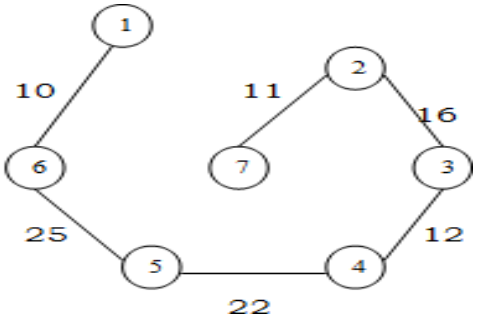
Stages in Prim's Algorithm

PRIM'S ALGORITHM: -

i) Select an edge with minimum cost and include in to the spanning tree.
ii) Among all the edges which are adjacent with the selected edge, select the one with minimum cost.
iii) Repeat step 2 until 'n' vertices and (n-1) edges are been included. And the sub graph obtained does not contain any cycles.

Notes: - At every state a decision is made about an edge of minimum cost to be included into the spanning tree. From the edges which are adjacent to the last edge included in the spanning tree i.e. at every stage the sub-graph obtained is a tree.





Prim's minimum spanning tree algorithm

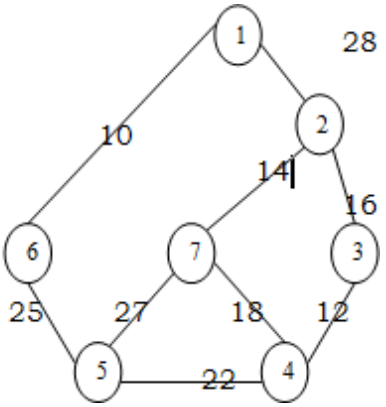
```
Algorithm Prim (E, cost, n,t)
// E is the set of edges in G. Cost (1:n, 1:n) is the
// Cost adjacency matrix of an n vertex graph such that
// Cost (i,j) is either a positive real no. or ∞ if no edge (i,j) exists.
//A minimum spanning tree is computed and
//Stored in the array T(1:n-1, 2).
//(t (i, 1), + t(i,2)) is an edge in the minimum cost spanning tree. The final cost is returned
        {
        Let (k, l) be an edge with min cost in E
        Min cost: = Cost (x,l);
        T(1,1):= k; + (1,2):= l;
for i:= 1 to n do//initialize near
        if (cost (i,l)<cost (i,k) then n east (i): l;
        else near (i): = k;
        near (k): = near (l): = 0;
        for i: = 2 to n-1 do
{//find n-2 additional edges for t
let j be an index such that near (i) ≠0 & cost (j, near (i)) is minimum;
t (i,1): = j + (i,2): = near (j);
min cost: = Min cost + cost (j, near (j));
near (j): = 0;
for k:=1 to n do // update near ()
if ((near (k) ≠0) and (cost {k, near (k)) > cost (k,j)))
then near Z(k): = ji
}
return mincost;
}
```

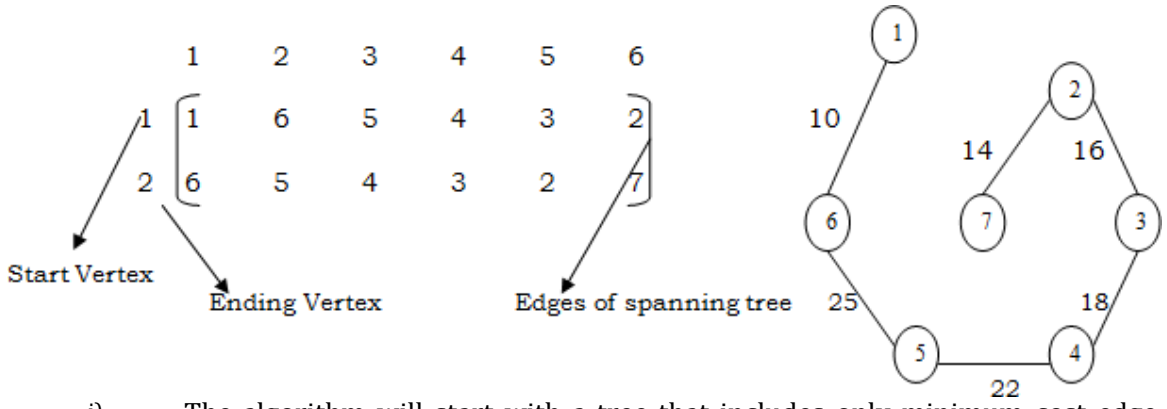
The algorithm takes four arguments E: set of edges, cost is nxn adjacency matrix cost of (i,j)= +ve integer, if an edge exists between i&j otherwise infinity. 'n' is no/: of vertices. 't' is a (n-1):2matrix which consists of the edges of spanning tree.

E = { (1,2), (1,6), (2,3), (3,4), (4,5), (4,7), (5,6), (5,7), (2,7) }

n = {1,2,3,4,5,6,7)

Cost	1	2	3	4	5	6	7
1	α	28	α	α	α	10	α
2	28	α	16	α	α	α	14
3	α	10	α	12	α	α	α
4	α	α	12	α	22	α	18
5	α	α	α	22	α	25	24
6	10	α	α	α	25	α	α
7	α	14	α	18	24	α	α





i) The algorithm will start with a tree that includes only minimum cost edge of G. Then edges are added to this tree one by one.

ii) The next edge (i,j) to be added is such that i is a vertex which is already included in the treed and j is a vertex not yet included in the tree and cost of i,j is minimum among all edges adjacent to 'i'.

iii) With each vertex 'j' next yet included in the tree, we assign a value near 'j'. The value near 'j' represents a vertex in the tree such that cost (j, near (j)) is minimum among all choices for near (j)

iv) We define near (j):= 0 for all the vertices 'j' that are already in the tree.

v) The next edge to include is defined by the vertex 'j' such that (near (j)) ≠ 0 and cost of (j, near (j)) is minimum.

Analysis: -

The time required by the prince algorithm is directly proportional to the no/: of vertices. If a graph 'G' has 'n' vertices then the time required by prim's algorithm is **$0(n^2)$**

Kruskal's Algorithm: Start with *no* nodes or edges in the spanning tree, and repeatedly add the cheapest edge that does not create a cycle.

In Kruskals algorithm for determining the spanning tree we arrange the edges in the increasing order of cost.

i) All the edges are considered one by one in that order and deleted from the graph and are included in to the spanning tree.

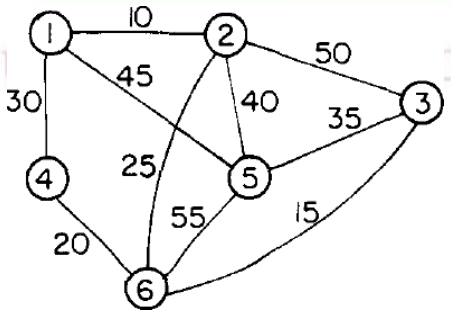
ii) At every stage an edge is included; the sub-graph at a stage need not be a tree. Infect it is a forest.

iii) At the end if we include 'n' vertices and n-1 edges without forming cycles then we get a single connected component without any cycles i.e. a tree with minimum cost.

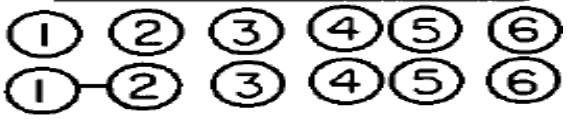
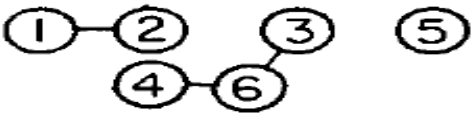
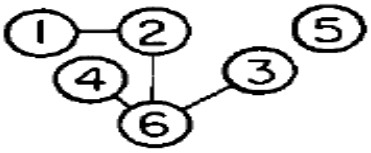
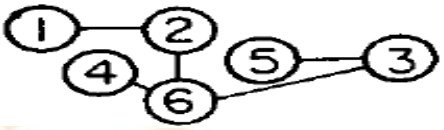
At every stage, as we include an edge in to the spanning tree, we get disconnected trees represented by various sets. While including an edge in to the spanning tree we need to check it does not form cycle. Inclusion of an edge (i,j) will form a cycle if i,j both are in same set. Otherwise the edge can be included into the spanning tree.

Kruskal minimum spanning tree algorithm

```
Algorithm Kruskal (E, cost, n,t)
//E is the set of edges in G. 'G' has 'n' vertices
//Cost {u,v} is the cost of edge (u,v) t is the set
//of edges in the minimum cost spanning tree
//The final cost is returned
{ construct a heap out of the edge costs using heapify;
        for i:= 1 to n do parent (i):= -1 // place in different sets
//each vertex is in different set                {1} {1} {3}
        i: = 0; min cost: = 0.0;
        While (i<n-1) and (heap not empty))do
{
Delete a minimum cost edge (u,v) from the heaps; and reheapify using adjust;
j:= find (u); k:=find (v);
if (j≠k) then
{  i: = 1+1;
   + (i,1)=u; + (i, 2)=v;
    mincost: = mincost+cost(u,v);
    Union (j,k);
    }
}
if (i≠n-1) then write ("No spanning tree");
   else return mincost;
}
```



Consider the above graph of , Using Kruskal's method the edges of this graph are considered for inclusion in the minimum cost spanning tree in the order (1, 2), (3, 6), (4, 6), (2, 6), (1, 4), (3, 5), (2, 5), (1, 5), (2, 3), and (5, 6). This corresponds to the cost sequence 10, 15, 20, 25, 30, 35, 40, 45, 50, 55. The first four edges are included in T. The next edge to be considered is (I, 4). This edge connects two vertices already connected in T and so it is rejected. Next, the edge (3, 5) is selected and that completes the spanning tree.

Edge	Cast	Spanning Forest
(1,2)	10	
(3,6)	15	
(4,6)	20	
(2,6)	25	
(1,4)	30	(reject)
(3,5)	35	

Stages in Kruskal's algorithm

Analysis: - If the no/: of edges in the graph is given by /E/ then the time for Kruskals algorithm is given by 0 (|E| log |E|).

Dynamic Programming

Dynamic programming is a name, coined by Richard Bellman in 1955. Dynamic programming, as greedy method, is a powerful algorithm design technique that can be used when the solution to the problem may be viewed as the result of a sequence of decisions. In the greedy method we make irrevocable decisions one at a time, using a greedy criterion. However, in dynamic programming we examine the decision sequence to see whether an optimal decision sequence contains optimal decision subsequence.

When optimal decision sequences contain optimal decision subsequences, we can establish recurrence equations, called *dynamic-programming recurrence equations*, that enable us to solve the problem in an efficient way.

Dynamic programming is based on the principle of optimality (also coined by Bellman). The principle of optimality states that no matter whatever the initial state and initial decision are, the remaining decision sequence must constitute an optimal decision sequence with regard to the state resulting from the first decision. The principle implies that an optimal decision sequence is comprised of optimal decision subsequences. Since the principle of optimality may not hold for some formulations of some problems, it is necessary to verify that it does hold for the problem being solved. Dynamic programming cannot be applied when this principle does not hold.

The steps in a dynamic programming solution are:

- Verify that the principle of optimality holds
- Set up the dynamic-programming recurrence equations
- Solve the dynamic-programming recurrence equations for the value of the optimal solution.
- Perform a trace back step in which the solution itself is constructed.

5.1 MULTI STAGE GRAPHS

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. In addition, if <u, v> is an edge in E, then u $\in$ Vi and v $\in$ Vi+1 for some i, 1 < i < k.

Let the vertex ‘s’ is the source, and ‘t’ 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 the costs of the edges on the path. The multistage graph problem is to find a minimum cost path from ‘s’ to ‘t’. Each set Vi defines a stage in the graph. Because of the constraints on E, every path from ‘s’ to ‘t’ starts in stage 1, goes to stage 2, then to stage 3, then to stage 4, and so on, and eventually terminates in stage k.

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 involves determining which vertex in v_{i+1}, $1 < i < k - 2$, is to be on the path. Let c (i, j) be the cost of the path from source to destination. Then using the forward approach, we obtain:

cost (i, j) = min {c (j, l) + cost (i + 1, l)}
l c Vi + 1
<j, l> c E

ALGORITHM:

Algorithm Fgraph (G, k, n, p)
// The input is a k-stage graph G = (V, E) with n vertices //
indexed in order or 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:
}
p [1] := 1; p [k] := n; // Find a minimum cost path.
for j := 2 to k - 1 do p [j] := d [p [j - 1]];}

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 j vertex in Vi. Let Bcost(i, j) be the cost of bp(i, j). From the backward approach we obtain:

Bcost (i, j) = min { Bcost (i –1, l) + c (l, j)}
l e Vi - 1
<l, j> e E

Algorithm Bgraph (G, k, n, p)
// Same function as Fgraph {
Bcost [1] := 0.0; for j := 2 to n do { / / C o m p u t e
B c o s t [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]];
}

Complexity Analysis:

The complexity analysis of the algorithm is fairly straightforward. Here, if G has ~E~ edges, then the time for the first for loop is CJ (V~ +~E).

EXAMPLE 1:

Find the minimum cost path from s to t in the multistage graph of five stages shown below. Do this first using forward approach and then using backward approach.

FORWARD APPROACH:

We use the following equation to find the minimum cost path from s to t: cost (i, j) = min {c (j, l) + cost (i + 1, l)}
l c Vi + 1
<j, l> c E

cost (1, 1) = min {c (1, 2) + cost (2, 2), c (1, 3) + cost (2, 3), c (1, 4) + cost (2, 4), c (1, 5) + cost (2, 5)}
= min {9 + cost (2, 2), 7 + cost (2, 3), 3 + cost (2, 4), 2 + cost (2, 5)}

Now first starting with,

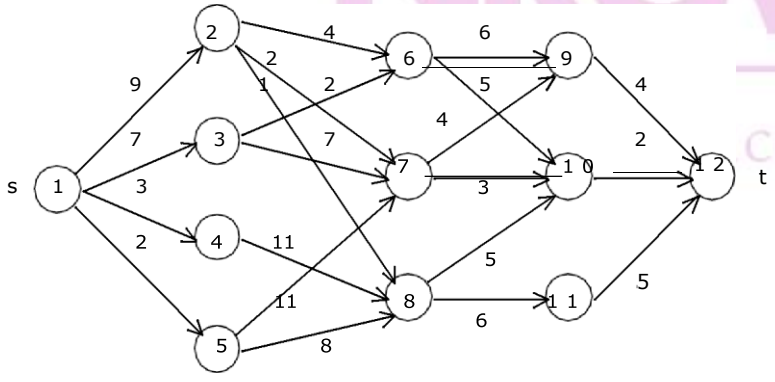
cost (2, 2) = min{c (2, 6) + cost (3, 6), c (2, 7) + cost (3, 7), c (2, 8) + cost (3, 8)} = min {4 + cost (3, 6), 2 + cost (3, 7), 1 + cost (3, 8)}

cost (3, 6) = min {c (6, 9) + cost (4, 9), c (6, 10) + cost (4, 10)}
= min {6 + cost (4, 9), 5 + cost (4, 10)}

cost (4, 9) = min {c (9, 12) + cost (5, 12)} = min {4 + 0) = 4 cost (4,

10) = min {c (10, 12) + cost (5, 12)} = 2

Therefore, cost (3, 6) = min {6 + 4, 5 + 2} = 7



cost (3, 7) = min {c (7, 9) + cost (4, 9) , c (7, 10) + cost (4, 10)}
= min {4 + cost (4, 9), 3 + cost (4, 10)}

cost (4, 9) = min {c (9, 12) + cost (5, 12)} = min {4 + 0} = 4 Cost (4,

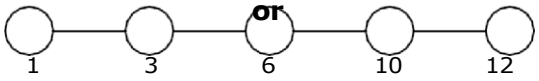
10) =

The path is

min

or

{c



(10,

2) + cost (5, 12)} = min {2 + 0} = 2 Therefore, cost (3, 7) = min {4 + 4, 3

+ 2} = min {8, 5} = 5

cost (3, 8) = min {c (8, 10) + cost (4, 10), c (8, 11) + cost (4, 11)}
= min {5 + cost (4, 10), 6 + cost (4 + 11)}

cost (4, 11) = min {c (11, 12) + cost (5, 12)} = 5

Therefore, cost (3, 8) = min {5 + 2, 6 + 5} = min {7, 11} = 7

Therefore, cost (2, 2) = min {4 + 7, 2 + 5, 1 + 7} = min {11, 7, 8} = 7

Therefore, cost (2, 3) = min {c (3, 6) + cost (3, 6), c (3, 7) + cost (3, 7)}
= min {2 + cost (3, 6), 7 + cost (3, 7)}
= min {2 + 7, 7 + 5} = min {9, 12} = 9

cost (2, 4) = min {c (4, 8) + cost (3, 8)} = min {11 + 7} = 18 cost (2, 5) =
min {c (5, 7) + cost (3, 7), c (5, 8) + cost (3, 8)} = min {11 + 5, 8 +
7} = min {16, 15} = 15

Therefore, cost (1, 1) = min {9 + 7, 7 + 9, 3 + 18, 2 + 15} = min
{16, 16, 21, 17} = 16

The minimum cost path is 16.

BACKWARD APPROACH:

We use the following equation to find the minimum cost path from t to s: Bcost (i, J) = min

{Bcost (i – 1, l) + c (l, J)}

l c vi – 1
<l, j> c E

Bcost (5, 12) = min {Bcost (4, 9) + c (9, 12), Bcost (4, 10) + c (10, 12),
Bcost (4, 11) + c (11, 12)}
= min {Bcost (4, 9) + 4, Bcost (4, 10) + 2, Bcost (4, 11) + 5}

Bcost (4, 9) = min {Bcost (3, 6) + c (6, 9), Bcost (3, 7) + c (7, 9)}
= min {Bcost (3, 6) + 6, Bcost (3, 7) + 4}

Bcost (3, 6) = min {Bcost (2, 2) + c (2, 6), Bcost (2, 3) + c (3, 6)}

= min {Bcost (2, 2) + 4, Bcost (2, 3) + 2}

Bcost (2, 2) = min {Bcost (1, 1) + c (1, 2)} = min {0 + 9} = 9 Bcost (2, 3) = min

{Bcost (1, 1) + c (1, 3)} = min {0 + 7} = 7 Bcost (3, 6) = min {9 + 4, 7 + 2} =

min {13, 9} = 9

Bcost (3, 7) = min {Bcost (2, 2) + c (2, 7), Bcost (2, 3) + c (3, 7), Bcost (2, 5) + c (5, 7)}

Bcost (2, 5) = min {Bcost (1, 1) + c (1, 5)} = 2

Bcost (3, 7) = min {9 + 2, 7 + 7, 2 + 11} = min {11, 14, 13} = 11 Bcost (4, 9) = min {9

+ 6, 11 + 4} = min {15, 15} = 15

Bcost (4, 10) = min {Bcost (3, 6) + c (6, 10), Bcost (3, 7) + c (7, 10), Bcost (3, 8) + c (8, 10)}

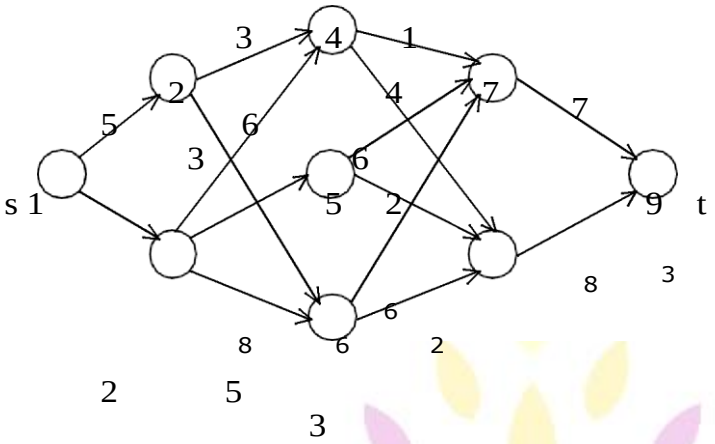
Bcost (3, 8) = min {Bcost (2, 2) + c (2, 8), Bcost (2, 4) + c (4, 8), Bcost (2, 5) + c (5, 8)}

Bcost (2, 4) = min {Bcost (1, 1) + c (1, 4)} = 3
Bcost (3, 8) = min {9 + 1, 3 + 11, 2 + 8} = min {10, 14, 10} = 10 Bcost (4, 10) = min {9 + 5, 11 + 3, 10 + 5} = min {14, 14, 15) = 14
Bcost (4, 11) = min {Bcost (3, 8) + c (8, 11)} = min {Bcost (3, 8) + 6} = min {10 + 6} = 16

Bcost (5, 12) = min {15 + 4, 14 + 2, 16 + 5} = min {19, 16, 21} = 16. **EXAMPLE**

2:

Find the minimum cost path from s to t in the multistage graph of five stages shown below. Do this first using forward approach and then using backward approach.



SOLUTION:

FORWARD APPROACH:

cost (i, J) = min {c (j, l) + cost (i + 1, l)}
l c Vi + 1
<J, l> EE

cost (1, 1) = min {c (1, 2) + cost (2, 2), c (1, 3) + cost (2, 3)}
= min {5 + cost (2, 2), 2 + cost (2, 3)}

cost (2, 2) = min {c (2, 4) + cost (3, 4), c (2, 6) + cost (3, 6)}
= min {3+ cost (3, 4), 3 + cost (3, 6)}

cost (3, 4) = min {c (4, 7) + cost (4, 7), c (4, 8) + cost (4, 8)}
= min {(1 + cost (4, 7), 4 + cost (4, 8)}

cost (4, 7) = min {c (7, 9) + cost (5, 9)} = min {7 + 0) = 7 cost (4, 8)

= min {c (8, 9) + cost (5, 9)} = 3

Therefore, cost (3, 4) = min {8, 7} = 7

cost (3, 6) = min {c (6, 7) + cost (4, 7), c (6, 8) + cost (4, 8)}
= min {6 + cost (4, 7), 2 + cost (4, 8)} = min {6 + 7, 2 + 3} = 5

Therefore, cost (2, 2) = min {10, 8} = 8

cost (2, 3) = min {c (3, 4) + cost (3, 4), c (3, 5) + cost (3, 5), c (3, 6) + cost (3,6)}

cost (3, 5) = min {c (5, 7) + cost (4, 7), c (5, 8) + cost (4, 8)}= min {6 + 7, 2 + 3} = 5

Therefore, cost (2, 3) = min {13, 10, 13} = 10

cost (1, 1) = min {5 + 8, 2 + 10} = min {13, 12} = 12

BACKWARD APPROACH:

Bcost (i, J) = min {Bcost (i – 1, l) = c (l, J)}
l E vi – 1
<l ,j>E E

Bcost (5, 9) = min {Bcost (4, 7) + c (7, 9), Bcost (4, 8) + c (8, 9)}
= min {Bcost (4, 7) + 7, Bcost (4, 8) + 3}

Bcost (4, 7) = min {Bcost (3, 4) + c (4, 7), Bcost (3, 5) + c (5, 7),
Bcost (3, 6) + c (6, 7)}
= min {Bcost (3, 4) + 1, Bcost (3, 5) + 6, Bcost (3, 6) + 6}

Bcost (3, 4) = min {Bcost (2, 2) + c (2, 4), Bcost (2, 3) + c (3, 4)}
= min {Bcost (2, 2) + 3, Bcost (2, 3) + 6}

Bcost (2, 2) = min {Bcost (1, 1) + c (1, 2)} = min {0 + 5} = 5

Bcost (2, 3) = min (Bcost (1, 1) + c (1, 3)} = min {0 + 2} = 2

Therefore, Bcost (3, 4) = min {5 + 3, 2 + 6} = min {8, 8} = 8

Bcost (3, 5) = min {Bcost (2, 3) + c (3, 5)} = min {2 + 5} = 7

Bcost (3, 6) = min {Bcost (2, 2) + c (2, 6), Bcost (2, 3) + c (3, 6)} = min
{5 + 5, 2 + 8} = 10

Therefore, Bcost (4, 7) = min {8 + 1, 7 + 6, 10 + 6} = 9

Bcost (4, 8) = min {Bcost (3, 4) + c (4, 8), Bcost (3, 5) + c (5, 8), Bcost
(3, 6) + c (6, 8)}
= min {8 + 4, 7 + 2, 10 + 2} = 9

Therefore, Bcost (5, 9) = min {9 + 7, 9 + 3} = 12 **All**

pairs shortest paths

In the all pairs shortest path problem, we are to find a shortest path between every pair of vertices in a directed graph G. That is, for every pair of vertices (i, j), we are to find a shortest path from i to j as well as one from j to i. These two paths are the same when G is undirected.

When no edge has a negative length, the all-pairs shortest path problem may be solved by using Dijkstra's greedy single source algorithm n times, once with each of the n vertices as the source vertex.

The all pairs shortest path problem is to determine a matrix A such that A (i, j) is the length of a shortest path from i to j. The matrix A can be obtained by solving n single-source

problems using the algorithm shortest Paths. Since each application of this procedure requires O (n^2) time, the matrix A can be obtained in O (n^3) time.

The dynamic programming solution, called Floyd's algorithm, runs in O (n^3) time. Floyd's algorithm works even when the graph has negative length edges (provided there are no negative length cycles).

The shortest i to j path in G, i ≠ j originates at vertex i and goes through some intermediate vertices (possibly none) and terminates at vertex j. If k is an intermediate vertex on this shortest path, then the subpaths from i to k and from k to j must be shortest paths from i to k and k to j, respectively. Otherwise, the i to j path is not of minimum length. So, the principle of optimality holds. Let A^k (i, j) represent the length of a shortest path from i to j going through no vertex of index greater than k, we obtain:

$$Ak(i, j) = \{\min \{\min_{1 \le k < n} \{A^{k-1}(i, k) + A^{k-1}(k, j)\}, c(i, j)\}$$

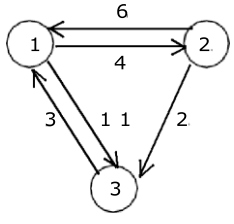
Algorithm All Paths (Cost, A, n)

```
// cost [1:n, 1:n] is the cost adjacency matrix of a graph which
// 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]);
}
```

Complexity Analysis: A Dynamic programming algorithm based on this recurrence involves in calculating n+1 matrices, each of size n x n. Therefore, the algorithm has a complexity of O (n^3).

Example 1:

Given a weighted digraph G = (V, E) with weight. Determine the length of the shortest path between all pairs of vertices in G. Here we assume that there are no cycles with zero or negative cost.



Cost adjacency matrix (A^0) =

$$\begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \sim & 0 \end{bmatrix}$$

General formula: $\min_{1<k<n} \{A^{k-1}(i, k) + A^{k-1}(k, j)\}, c(i, j)\}$

Solve the problem for different values of k = 1, 2

and 3 **Step 1**: Solving the equation for, k = 1;

A1 (1, 1) = min {(A^o (1, 1) + A^o (1, 1)), c (1, 1)} = min {0 + 0, 0} = 0 A1 (1, 2) = min {(A^o (1, 1) + A^o (1, 2)), c (1, 2)} = min {(0 + 4), 4} = 4

A1 (1, 3) = min {(A^o (1, 1) + A^o (1, 3)), c (1, 3)} = min {(0 + 11), 11} = 11 A1 (2, 1) = min {(A^o (2, 1) + A^o (1, 1)), c (2, 1)} = min {(6 + 0), 6} = 6

A1 (2, 2) = min {(A^o (2, 1) + A^o (1, 2)), c (2, 2)} = min {(6 + 4), 0)} = 0 A1 (2, 3) = min {(A^o (2, 1) + A^o (1, 3)), c (2, 3)} = min {(6 + 11), 2} = 2 A1 (3, 1) = min {(A^o (3, 1) + A^o (1, 1)), c (3, 1)} = min {(3 + 0), 3} = 3 A1 (3, 2) = min {(A^o (3, 1) + A^o (1, 2)), c (3, 2)} = min {(3 + 4), oc} = 7 A1 (3, 3) = min {(A^o (3, 1) + A^o (1, 3)), c (3, 3)} = min {(3 + 11), 0} = 0

$A_{(1)}$ = ~0 4 11□
~ ~
~6 0 2~
~L3 7 0~U

Step 2: Solving the equation for, K = 2;

A_2 (1, 1) = min {(A^1 (1, 2) + A^1 (2, 1), c (1, 1)} = min {(4 + 6), 0} + A^1 = 0
A_2 (1, 2) = min {(A^1 (1, 2) (2, 2), c (1, 2)} = min {(4 + 0), 4} + A^1 (2, = 4
A_2 (1, 3) = min {(A^1 (1, 2) 3), c (1, 3)} = min {(4 + 2), 11} = 6
A_2 (2, 1) = min {(A (2, 2) + A (2, 1), c (2, 1)} = min {(0 + 6), 6} = 6
A_2 (2, 2) = min {(A (2, 2) + A (2, 2), c (2, 2)} = min {(0 + 0), 0} = 0
A_2 (2, 3) = min {(A (2, 2) + A (2, 3), c (2, 3)} = min {(0 + 2), 2} = 2
A_2 (3, 1) = min {(A (3, 2) + A (2, 1), c (3, 1)} = min {(7 + 6), 3} = 3
A_2 (3, 2) = min {(A (3, 2) + A (2, 2), c (3, 2)} = min {(7 + 0), 7} = 7
3) = min {(A (3, 2) + A (2, 3), c (3, 3)} = min {(7 + 2), 0} = 0

$A_{(2)}$ = ~0 4 6 1
~ 2~
~6 0 ~
~L3 7 0~~

Step 3: Solving the equation for, k = 3;

A3 (1, 1) = min {A^2 (1, 3) + A^2 (3, 1), c (1, 1)} = min {(6 + 3), 0} = 0

A3 (1, 2) = min {A^2 (1, 3) + A^2 (3, 2), c (1, 2)} = min {(6 + 7), 4} = 4

A3 (1, 3) = min {A^2 (1, 3) + A^2 (3, 3), c (1, 3)} = min {(6 + 0), 6} = 6

A3 (2, 1) = min {A^2 (2, 3) + A^2 (3, 1), c (2, 1)} = min {(2 + 3), 6} = 5

A3 (2, 2) = min {A^2 (2, 3) + A^2 (3, 2), c (2, 2)} = min {(2 + 7), 0} = 0

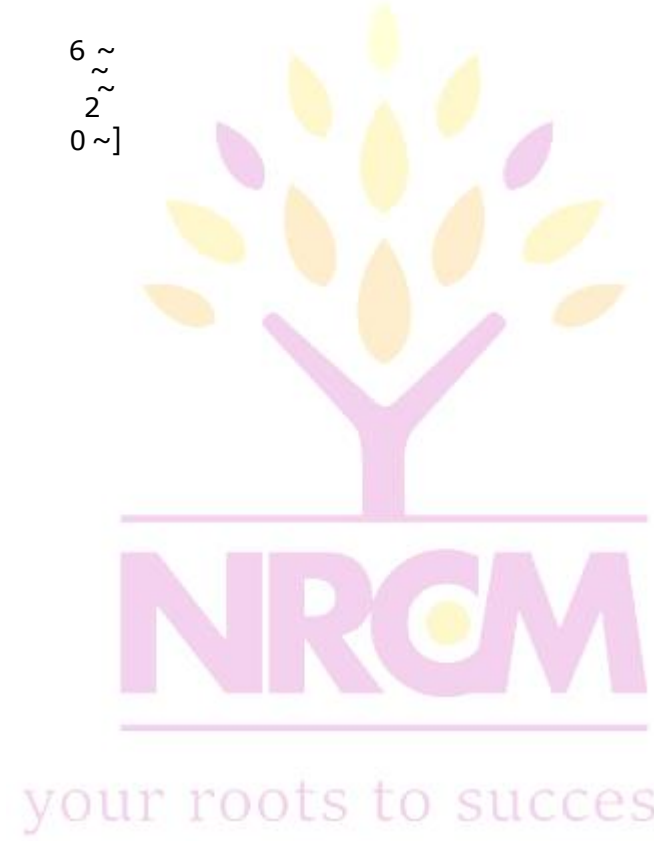
A3 (2, 3) = min {A^2 (2, 3) + A^2 (3, 3), c (2, 3)} = min {(2 + 0), 2} = 2

A3 (3, 1) = min {A^2 (3, 3) + A^2 (3, 1), c (3, 1)} = min {(0 + 3), 3} = 3

A3 (3, 2) = min {A^2 (3, 3) + A^2 (3, 2), c (3, 2)} = min {(0 + 7), 7} = 7

A3 (3, 3) = min {A^2 (3, 3) + A^2 (3, 3), c (3, 3)} = min {(0 + 0), 0} = 0

$A_{(3)}$ = ~ 0 4 6 ~
0 ~
2 ~
~5~3 7 0 ~]



TRAVELLING SALESPERSON PROBLEM

Let G = (V, E) be a directed graph with edge costs Cij. The variable cij is defined such that cij > 0 for all I and $_j$ and cij = a if < i, j> o E. Let |V| = n and assume n > 1. A tour of G is a directed simple cycle that includes every vertex in V. The cost of a tour is the sum of the cost of the edges on the tour. The traveling sales person problem is to find a tour of minimum cost. The tour is to be a simple path that starts and ends at vertex 1.

Let g (i, S) be the length of 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:

$$g(1, V - \{1\}) = 2 \sim k \sim n \sim c1k \sim g \sim k, V \sim \sim 1, k \sim\sim \quad \text{-- 1}$$

$$\min$$

$$\text{-- 2}$$

Generalizing equation 1, we obtain (for i o S)

$$g(i, S) = \min\{ci\, j \quad j \in S$$

The Equation can be solved for g (1, V – 1}) if we know g (k, V – {1, k}) for all choices of k.

Complexity Analysis:

For each value of |S| there $+ g\ (i, S - (j))$ are n – 1 choices for i. The number of distinct sets S of

size k not including 1 and i is $I\, k \sim \sim n\text{-}2 \sim$

Hence, the total number of g (i, S)'s to be computed before computing g (1, V – {1}) is:

$$\sim n\text{-}2\sim \quad \sim \sim n \sim 1 \sim \sim \quad \sim k \sim \quad k \sim 0$$

To calculate this sum, we use the binominal theorem:

$$n-1 \quad [((n - 2)\ ((n - 2)\ ((n - 2) \quad ((n - 2)1 \quad (n–1)111 \quad 11+\mathbf{i} \quad iI+ii \quad iI+\text{---}\sim\sim\sim \quad \sim \quad \sim\sim 0 \quad) \sim 1\ ) \sim 2\ ) \quad \sim(n\sim^2)\sim\sim$$

According to the binominal theorem:

$$[((n - 2)\ ((n - 2)\ ((n - 2 \quad ((n - 2)1 \quad il \quad 11+\mathbf{i} \quad iI+\mathbf{i} \quad \sim\sim\sim\sim\sim\sim\sim\sim \quad \sim\sim\sim=2n\text{-}2 \quad \sim\sim 0 \quad \sim \sim 1 \sim \sim 2 \sim \quad \sim(n - 2))]$$

Therefore,

$$n-1 \quad \sim n\_2' \quad \sim (n \_ 1 \sim \sim \sim k = (n - 1)\ _2 n \sim 2$$

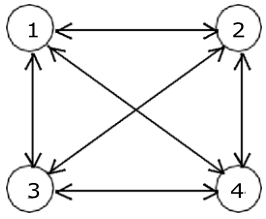
This is Φ (n 2^{n-2}), so there are exponential number of calculate. Calculating one g (i, S) require finding the minimum of at most n quantities. Therefore, the entire algorithm is

Φ (n^2 2^{n-2}). This is better than enumerating all n! different tours to find the best one. So, we have traded on exponential growth for a much smaller exponential growth.

The most serious drawback of this dynamic programming solution is the space needed, which is O (n 2^n). This is too large even for modest values of n.

Example 1:

For the following graph find minimum cost tour for the traveling salesperson problem:



The cost adjacency matrix =

0	10	15	20
5	0	9	10
6	13	0	12
8	9	0	0

Let us start the tour from vertex 1:

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

More generally writing:

$$g(i, s) = \min \{cij + g(J, s - \{J\})\} \quad - \quad (2)$$

Clearly, g (i, T) = ci1 , $1 \le i \le n$. So,

g (2, T) = C21 = 5

g (3, T) = C31 = 6

g (4, ~) = C41 = 8

Using equation – (2) we obtain:

g (1,{2, 3, 4}) = min {c12 + g (2, {3, 4}, c13 + g (3, {2, 4}), c14 + g (4, {2, 3})}

g (2, {3, 4}) = min {c23 + g (3, {4}), c24 + g (4, {3})}
= min {9 + g (3, {4}), 10 + g (4, {3})}

g (3, {4}) = min {c34 + g (4, T)} = 12 + 8 = 20

g (4, {3}) = min {c43 + g (3, ~)} = 9 + 6 = 15

Therefore, g (2, {3, 4}) = min {9 + 20, 10 + 15} = min {29, 25} = 25

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

g (2, {4}) = min {c24 + g (4, T)} = 10 + 8 = 18

g (4, {2}) = min {c42 + g (2, ~)} = 8 + 5 = 13

Therefore, g (3, {2, 4}) = min {13 + 18, 12 + 13} = min {41, 25} = 25

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

g (2, {3}) = min {c23 + g (3, ~} = 9+ 6 = 15

g (3, {2}) = min {c32 + g (2, T} = 13+ 5 = 18

Therefore, g (4, {2, 3}) = min {8 + 15, 9 + 18} = min {23, 27} = 23

g (1, {2, 3, 4}) = min {c12 + g (2, {3, 4}), c13 + g (3, {2, 4}), c14 + g (4, {2, 3})} = min {10 + 25, 15 + 25, 20 + 23} = min {35, 40, 43} = 35

The optimal tour for the graph has length = 35 The

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

OPTIMAL BINARY SEARCH TREE

Let us assume that the given set of identifiers is {a1, . . . , an} with a1 < a2 < < an. Let p (i) be the probability with which we search for ai. Let q (i) be the probability that the identifier x being searched for is such that ai < x < ai+1, 0 < i < n (assume a0 = - ~ and an+1 = +oc). We have to arrange the identifiers in a binary search tree in a way that minimizes the expected total access time.

In a binary search tree, the number of comparisons needed to access an element at depth 'd' is d + 1, so if 'ai' is placed at depth 'di', then we want to minimize:

$$\sum_{i=1}^{n} Pi\ (1 + di\ ) .$$

Let P (i) be the probability with which we shall be searching for 'ai'. Let Q (i) be the probability of an un-successful search. Every internal node represents a point where a successful search may terminate. Every external node represents a point where an unsuccessful search may terminate.

The expected cost contribution for the internal node for 'ai' is:

$$P\ (i) * level\ (ai\ ) .$$

Unsuccessful search terminate with I = 0 (i.e at an external node). Hence the cost contribution for this node is:

Q (i) * level ((Ei) - 1)

The expected cost of binary search tree is:

$$\sum_{i=1}^{n} P(i) * level(a_i) + \sum_{i=0}^{n} Q(i) * level((E_i) - 1)$$

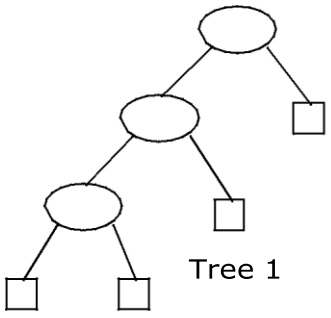
Given a fixed set of identifiers, we wish to create a binary search tree organization. We may expect different binary search trees for the same identifier set to have different performance characteristics.

The computation of each of these c(i, j)'s requires us to find the minimum of 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 therefore $O(nm - m^2)$.

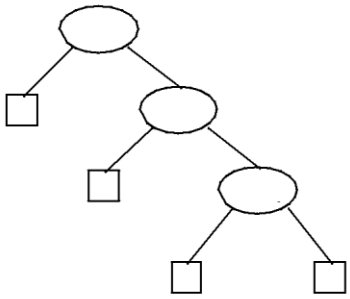
The total time to evaluate all the c(i, j)'s and r(i, j)'s is therefore:

$$\sum (nm - m^2) = O(n^3) \quad 1 < m < n$$

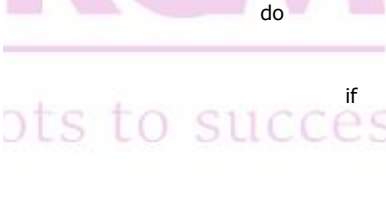
Example 1: The possible binary search trees for the identifier set (a1, a2, a3) = (do, if, stop) are as follows. Given the equal probabilities p (i) = Q (i) = 1/7 for all i, we have:



Tree 1



Tree 2



Tree 3

Cost (tree # 1) = $\left(\frac{1}{7} \times 1 + \frac{1}{7} \times 2 + \frac{1}{7} \times 3\right) + \left(\frac{1}{7} \times 1\right.$

1 + 2 + 3 1 + 2 + 3 + 3 6 + 9 15

Cost (tree # 3) = ~ 1 x 1 + 1 x 2 + 1 x 3~ ~ + (~ 1 x 1 + 1 x 2 + 1 x 3 + 1 x 3~ ~
~ 7 7 7) ~ 7 7 7 7)

= 1 + 2 + 3 1 + 2 + 3 + 3 6 + 9 15
7 + 7 ~ ~

Cost (tree # 4) = ~ 1 x 1 + 1 x 2 ~ 1 x 3~ ~ ~ (1 x 1 + 1 x 2 + 1 x 3 + 1 x 3~ ~
~ 7 7 7) ~ 7 7 7 7)

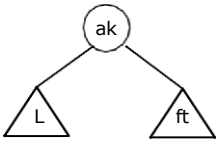
= 1 + 2 + 3 1 + 2 + 3 + 3 6 + 9 15
7

Cost (tree # 2) = (1 x 1 + 1 1 x 2~ (~ 1x2+ 1 x 2 + 1 x 2 + 1 x2~
~ 7 7 x2+ 7) + ~ 7 7 7 7)

= 1 + 2 + 2 2 + 2 + 2 + 2 5 + 8 13
7 + 7 ~ 7 ~ 7

Huffman coding tree solved by a greedy algorithm has a limitation of having the data only at the leaves and it must not preserve the property that all nodes to the left of the root have keys, which are less etc. Construction of an optimal binary search tree is harder, because the data is not constrained to appear only at the leaves, and also because the tree must satisfy the binary search tree property and it must preserve the property that all nodes to the left of the root have keys, which are less.

A dynamic programming solution to the problem of obtaining an optimal binary search tree can be viewed by constructing a tree as a result of sequence of decisions by holding the principle of optimality. A possible approach to this is to make a decision as which of the ai's be arraigned to the root node at 'T'. If we choose 'ak' then is clear that the internal nodes for a1, a2, ... ak-1 as well as the external nodes for the classes Eo, E1, ... Ek-1 will lie in the left sub tree, L, of the root. The remaining nodes will be in the right subtree, ft. The structure of an optimal binary search tree is:



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

$$\text{Cost (ft)} = \sum_{i=K}^{n} P(i) * level(a_i) + \sum_{i=K}^{n} Q(i) * level(E_i) - 1$$

The C (i, J) can be computed as:

$$C(i, J) = \min_{i<k<J} \{C(i, k-1) + C(k, J) + P(K) + w(i, K-1) + w(K, J)\}$$

$$= \min_{i<k<J} \{C(i, K-1) + C(K, J)\} + w(i, J) \quad -- \quad (1)$$

Where $W(i, J) = P(J) + Q(J) + w(i, J-1)$ -- (2)

Initially C (i, i) = 0 and w (i, i) = Q (i) for $0 < i < n$.

Equation (1) may be solved for C (0, n) by first computing all C (i, J) such that J - i = 1 Next, we can compute all C (i, J) such that J - i = 2, Then all C (i, J) with J - i = 3 and so on.
C (i, J) is the cost of the optimal binary search tree 'Tij' during computation we record the root R (i, J) of each tree 'Tij'. Then an optimal binary search tree may be constructed from these R (i, J). R (i, J) is the value of 'K' that minimizes equation (1).

We solve the problem by knowing W (i, i+1), C (i, i+1) and R (i, i+1), $0 \leq i < 4$;

Knowing W (i, i+2), C (i, i+2) and R (i, i+2), $0 \leq i < 3$ and repeating until W (0, n), C (0, n) and R (0, n) are obtained.

The results are tabulated to recover the actual tree.

Example 1:

Let n = 4, and (a1, a2, a3, a4) = (do, if, need, while) Let P (1: 4) = (3, 3, 1, 1) and Q (0: 4) = (2, 3, 1, 1, 1)

Solution:

Table for recording W (i, j), C (i, j) and R (i, j):

Column Row	**0**	**1**	**2**	**3**	**4**
0	2, 0, 0	3, 0, 0	1, 0, 0	1, 0, 0,	1, 0, 0
1	8, 8, 1	7, 7, 2	3, 3, 3	3, 3, 4	
2	12, 19, 1	9, 12, 2	5, 8, 3		
3	14, 25, 2	11, 19, 2			
4	16, 32, 2				

This computation is carried out row-wise from row 0 to row 4. Initially, W (i, i) = Q (i) and C (i, i) = 0 and R (i, i) = 0, $0 < i < 4$.

Solving for C (0, n):

First, computing all C (i, j) such that j - i = 1; j = i + 1 and as 0 < i < 4; i = 0, 1, 2 and 3; i < k ≤ J. Start with i = 0; so j = 1; as i < k ≤ j, so the possible value for k = 1

W (0, 1) = P (1) + Q (1) + W (0, 0) = 3 + 3 + 2 = 8
C (0, 1) = W (0, 1) + min {C (0, 0) + C (1, 1)} = 8
R (0, 1) = 1 (value of 'K' that is minimum in the above equation).

Next with i = 1; so j = 2; as i < k ≤ j, so the possible value for k = 2

W (1, 2) = P (2) + Q (2) + W (1, 1) = 3 + 1 + 3 = 7
C (1, 2) = W (1, 2) + min {C (1, 1) + C (2, 2)} = 7
R (1, 2) = 2

Next with i = 2; so j = 3; as i < k ≤ j, so the possible value for k = 3

W (2, 3) = P (3) + Q (3) + W (2, 2) = 1 + 1 + 1 = 3

C (2, 3) = W (2, 3) + min {C (2, 2) + C (3, 3)} = 3 + [(0 + 0)] = 3
ft (2, 3) = 3

Next with i = 3; so j = 4; as i < k ≤ j, so the possible value for k = 4

W (3, 4) = P (4) + Q (4) + W (3, 3) = 1 + 1 + 1 = 3
C (3, 4) = W (3, 4) + min {[C (3, 3) + C (4, 4)]} = 3 + [(0 + 0)] = 3
ft (3, 4) = 4

Second, Computing all C (i, j) such that j - i = 2; j = i + 2 and as 0 < i < 3; i = 0, 1, 2; i < k ≤ J. Start with i = 0; so j = 2; as i < k ≤ J, so the possible values for k = 1 and 2.

W (0, 2) = P (2) + Q (2) + W (0, 1) = 3 + 1 + 8 = 12
C (0, 2) = W (0, 2) + min {(C (0, 0) + C (1, 2)), (C (0, 1) + C (2, 2))} = 12
+ min {(0 + 7, 8 + 0)} = 19
ft (0, 2) = 1
Next, with i = 1; so j = 3; as i < k ≤ j, so the possible value for k = 2 and 3.
W (1, 3) = P (3) + Q (3) + W (1, 2) = 1 + 1+ 7 = 9
C (1, 3) = W (1, 3) + min {[C (1, 1) + C (2, 3)], [C (1, 3) + C (3, 3)]}
= W (1, 2) + min {(0 + 3), (7 + 0)} = 9 + 3 = 12
ft (1, 3) = 2

Next, with i = 2; so j = 4; as i < k ≤ j, so the possible value for k = 3 and 4.

W (2, 4) = P (4) + Q (4) + W (2, 3) = 1 + 1 + 3 = 5
C (2, 4) = W (2, 4) + min {[C (2, 2) + C (3, 4)], [C (2, 3) + C (4, 4)]
= 5 + min {(0 + 3), (3 + 0)} = 5 + 3 = 8
ft (2, 4) = 3

Third, Computing all C (i, j) such that J - i = 3; j = i + 3 and as 0 < i < 2; i = 0, 1; i < k ≤ J. Start with i = 0; so j = 3; as i < k ≤ j, so the possible values for k = 1, 2 and 3.

W (0, 3) = P (3) + Q (3) + W (0, 2) = 1 + 1 + 12 = 14
C (0, 3) W (0, 3) + min {[C (0, 0) + C (1, 3)], [C (0, 1) + C (2, 3)],
[C (0, 2) + C (3, 3)]}
+ 11 = 25
ft (0, 3) 14 + min {(0 + 12), (8 + 3), (19 + 0)} = 14

Start with i = 1; so j = 4; as $i < k \leq j$, so the possible values for k = 2, 3 and 4.

W (1, 4) = P (4) + Q (4) + W (1, 3) = 1 + 1 + 9 = 11 = W 2)
C (1, 4) (1, 4) + min {[C (1, 1) + C (2, 4)], [C (1, + C (3, 4)],
[C (1, 3) + C (4, 4)]}
ft (1, 4) = 11 + min {(0 + 8), (7 + 3), (12 + 0)} = 11 = 2 + 8 = 19

Fourth, Computing all C (i, j) such that j - i = 4; j = i + 4 and as 0 < i < 1; i = 0; i < k ≤ J.

Start with i = 0; so j = 4; as i < k ≤ j, so the possible values for k = 1, 2, 3 and 4.

W (0, 4) = P (4) + Q (4) + W (0, 3) = 1 + 1 + 14 = 16

C (0, 4) = W (0, 4) + min {[C (0, 0) + C (1, 4)], [C (0, 1) + C (2, 4)],
[C (0, 2) + C (3, 4)], [C (0, 3) + C (4, 4)]}
= 16 + min [0 + 19, 8 + 8, 19+3, 25+0] = 16 + 16 = 32 ft (0, 4) = 2

From the table we see that C (0, 4) = 32 is the minimum cost of a binary search tree for (a1, a2, a3, a4). The root of the tree 'T04' is 'a2'.

Hence the left sub tree is 'T01' and right sub tree is T24. The root of 'T01' is 'a1' and the root of 'T24' is a3.

The left and right sub trees for 'T01' are 'T00' and 'T11' respectively. The root of T01 is 'a1'

The left and right sub trees for T24 are T22 and T34 respectively.

The root of T24 is 'a3'.

The root of T22 is null

The root of T34 is a4.

Example 2:

Consider four elements a1, a2, a3 and a4 with Q0 = 1/8, Q1 = 3/16, Q2 = Q3 = Q4 = 1/16 and p1 = 1/4, p2 = 1/8, p3 = p4 =1/16. Construct an optimal binary search tree. Solving for C (0, n):

First, computing all C (i, j) such that j - i = 1; j = i + 1 and as 0 < i < 4; i = 0, 1, 2 and 3; i < k ≤ J. Start with i = 0; so j = 1; as i < k ≤ j, so the possible value for k = 1

W (0, 1) = P (1) + Q (1) + W (0, 0) = 4 + 3 + 2 = 9

C (0, 1) = W (0, 1) + min {C (0, 0) + C (1, 1)} = 9 + [(0 + 0)] = 9 ft (0, 1) = 1 (value of 'K' that is minimum in the above equation).

Next with i = 1; so j = 2; as i < k ≤ j, so the possible value for k = 2

W (1, 2) = P (2) + Q (2) + W (1, 1) = 2 + 1 + 3 = 6
C (1, 2) = W (1, 2) + min {C (1, 1) + C (2, 2)} = 6 + [(0 + 0)] = 6 ft (1, 2) = 2

Next with i = 2; so j = 3; as i < k ≤ j, so the possible value for k = 3

W (2, 3) = P (3) + Q (3) + W (2, 2) = 1 + 1 + 1 = 3
C (2, 3) = W (2, 3) + min {C (2, 2) + C (3, 3)} = 3 + [(0 + 0)] = 3

ft (2, 3) = 3

Next with i = 3; so j = 4; as i < k ≤ j, so the possible value for k = 4

W (3, 4) = P (4) + Q (4) + W (3, 3) = 1 + 1 + 1 = 3
C (3, 4) = W (3, 4) + min {[C (3, 3) + C (4, 4)]} = 3 + [(0 + 0)] = 3
ft (3, 4) = 4

Second, Computing all C (i, j) such that j - i = 2; j = i + 2 and as 0 < i < 3; i = 0, 1, 2; i < k ≤ J

Start with i = 0; so j = 2; as i < k ≤ j, so the possible values for k = 1 and 2.

W (0, 2) = P (2) + Q (2) + W (0, 1) = 2 + 1 + 9 = 12
C (0, 2) = W (0, 2) + min {(C (0, 0) + C (1, 2)), (C (0, 1) + C (2, 2))} = 12 + min {(0 + 6, 9 + 0)} = 12 + 6 = 18
ft (0, 2) = 1
Next, with i = 1; so j = 3; as i < k ≤ j, so the possible value for k = 2 and 3.
W (1, 3) = P (3) + Q (3) + W (1, 2) = 1 + 1+ 6 = 8
C (1, 3) = W (1, 3) + min {[C (1, 1) + C (2, 3)], [C (1, 2) + C (3, 3)]}
= W (1, 3) + min {(0 + 3), (6 + 0)} = 8 + 3 = 11
ft (1, 3) = 2

Next, with i = 2; so j = 4; as i < k ≤ j, so the possible value for k = 3 and 4.

W (2, 4) = P (4) + Q (4) + W (2, 3) = 1 + 1 + 3 = 5
C (2, 4) = W (2, 4) + min {[C (2, 2) + C (3, 4)], [C (2, 3) + C (4, 4)]
= 5 + min {(0 + 3), (3 + 0)} = 5 + 3 = 8
ft (2, 4) = 3

Third, Computing all C (i, j) such that J - i = 3; j = i + 3 and as 0 < i < 2; i = 0, 1; i < k ≤ J. Start with i = 0; so j = 3; as i < k ≤ j, so the possible values for k = 1, 2 and 3.

W (0, 3) = P (3) + Q (3) + W (0, 2) = 1 + 1 + 12 = 14
C (0, 3) = W (0, 3) + min {[C (0, 0) + C (1, 3)], [C (0, 1) + C (2, 3)], [C (0, 2) + C (3, 3)]}
= 14 + min {(0 + 11), (9 + 3), (18 + 0)} = 14 + 11 = 25 ft (0, 3) = 1

Start with i = 1; so j = 4; as i < k ≤ j, so the possible values for k = 2, 3 and 4.

W (1, 4) = P (4) + Q (4) + W (1, 3) = 1 + 1 + 8 = 10 = W (1, 2)
C (1, 4) (1, 4) + min {[C (1, 1) + C (2, 4)], [C (1, 3) + C (4, 4)]}
= 10 + min {(0 + 8), (6 + 3), (11 + 0)} = 10 = 2 + C (3, 4)], + 8 = 18
ft (1, 4)

Fourth, Computing all C (i, j) such that J - i = 4; j = i + 4 and as 0 < i < 1; i = 0; i < k ≤ J. Start with i = 0; so j = 4; as i < k ≤ j, so the possible values for k = 1, 2, 3 and 4.

W (0, 4) = P (4) + Q (4) + W (0, 3) = 1 + 1 + 14 = 16
C (0, 4) = W (0, 4) + min {[C (0, 0) + C (1, 4)], [C (0, 1) + C (2, 4)], [C (0, 2) + C (3, 4)], [C (0, 3) + C (4, 4)]}

= 16 + min [0 + 18, 9 + 8, 18 + 3, 25 + 0] = 16 + 17 = 33 R (0, 4)
= 2

Table for recording W (i, j), C (i, j) and R (i, j)

Column Row	**0**	**1**	**2**	**3**	**4**
0	2, 0, 0	1, 0, 0	1, 0, 0	1, 0, 0,	1, 0, 0
1	9, 9, 1	6, 6, 2	3, 3, 3	3, 3, 4	
2	12, 18, 1	8, 11, 2	5, 8, 3		
3	14, 25, 2	11, 18, 2			
4	16, 33, 2				

From the table we see that C (0, 4) = 33 is the minimum cost of a binary search tree for (a1, a2, a3, a4)

The root of the tree 'T04' is 'a2'.

Hence the left sub tree is 'T01' and right sub tree is T24. The root of 'T01' is 'a1' and the root of 'T24' is a3.

The left and right sub trees for 'T01' are 'T00' and 'T11' respectively. The root of T01 is 'a1'

The left and right sub trees for T24 are T22 and T34 respectively.

The root of T24 is 'a3'.

The root of T22 is null.

The root of T34 is a4.

0/1 – KNAPSACK

We are given n objects and a knapsack. Each object i has a positive weight wi and a positive value Vi. The knapsack can carry a weight not exceeding W. Fill the knapsack so that the value of objects in the knapsack is optimized.

A solution to the knapsack problem can be obtained by making a sequence of decisions on the variables x1, x2, ... , xn. A decision on variable xi involves determining which of the values 0 or 1 is to be assigned to it. Let us assume that

decisions on the xi are made in the order xn, xn-1,.....x1. Following a decision on xn, we may be in one of two possible states: the capacity remaining in m – wn and a profit of pn has accrued. It is clear that the remaining decisions xn-1, ... , x1 must be optimal with respect to the problem state resulting from the decision on xn. Otherwise, xn,. . . . , x1 will not be optimal. Hence, the principal of optimality holds.

$$Fn(m) = \max \{fn-1(m), fn-1(m - wn) + pn\} \quad -- \quad 1$$

For arbitrary fi (y), i > 0, this equation generalizes to:

$$Fi(y) = \max \{fi-1(y), fi-1(y - wi) + pi\} \quad – \quad 2$$

Equation-2 can be solved for fn (m) by beginning with the knowledge fo (y) = 0 for all y and fi (y) = - ~, y < 0. Then f1, f2, . . . fn can be successively computed using equation–2.

When the wi's are integer, we need to compute fi (y) for integer y, 0 < y < m. Since fi (y) = - ~ for y < 0, these function values need not be computed explicitly. Since each fi can be computed from fi - 1 in **Θ** (m) time, it takes **Θ** (m n) time to compute fn. When the wi's are real numbers, fi (y) is needed for real numbers y such that 0 < y < m. So, fi cannot be explicitly computed for all y in this range. Even when the wi's are integer, the explicit **Θ** (m n) computation of fn may not be the most efficient computation. So, we explore **an alternative method for both cases.**

The fi (y) is an ascending step function; i.e., there are a finite number of y's, 0 = y1 < y2 < . . . . < yk, such that fi (y1) < fi (y2) <.....< fi (yk); fi (y) = - ~ , y < y1; fi (y) = f (yk), y > yk; and fi (y) = fi (yj), yj < y < yj+1. So, we need to compute only fi (yj), 1 < j < k. We use the ordered set S^i = {(f (yj), yj) | 1 < j < k} to represent fi (y). Each number of S^i is a pair (P, W), where P = fi (yj) and W = yj. Notice that S^0 = {(0, 0)}. We can compute S^{i+1} from Si by first computing:

$$Si\ 1 = \{(P, W) \mid (P – pi, W – wi)\ e\ S^i\}$$

Now, S^{i+1} can be computed by merging the pairs in S^i and Si 1 together. Note that if Si+1 contains two pairs (Pj, Wj) and (Pk, Wk) with the property that Pj < Pk and Wj > Wk, then the pair (Pj, Wj) can be discarded because of equation-2. Discarding or purging rules such as this one are also known as dominance rules. Dominated tuples get purged. In the above, (Pk, Wk) dominates (Pj, Wj).

Reliability Design

The problem is to design a system that is composed of several devices connected in series. Let ri be the reliability of device Di (that is ri is the probability that device i will function properly) then the reliability of the entire system is fT ri. Even if the individual devices are very reliable (the ri's are very close to one), the reliability of the system may not be very good. For example, if n = 10 and ri = 0.99, i < i < 10, then fT ri = .904. Hence, it is desirable to duplicate devices. Multiply copies of the same device type are connected in parallel.

If stage i contains mi copies of device Di. Then the probability that all mi have a malfunction is (1 - ri) mi. Hence the reliability of stage i becomes $1 - (1 - r)^{mi}$.

i

The reliability of stage 'i' is given by a function ~i (mi).

Our problem is to use device duplication. This maximization is to be carried out under a cost constraint. Let ci 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:

Maximize ~ *qi* (*mi* ~

$1 \le i \le n$

Subject to ~ *Ci mi* < *C*

$1 \le i \le n$

mi > 1 and interger, 1 < i < n

Assume each Ci > 0, each mi must be in the range 1 < mi < ui, where

$$ui = \left\lfloor C + Ci - \sum_{1 \le j \le n} Cj \Big/ Ci \right\rfloor$$

The upper bound ui follows from the observation that mj > 1

An optimal solution m1, m2 ..... mn is the result of a sequence of decisions, one decision for each mi.

Let fi (x) represent the maximum value of $\sum_{1 \le j \le i} q(mj)$

Subject to the constrains:

$$\sum_{1 \le j \le i} Cj\, mj \le x \quad \text{and } 1 < mj < uj,\ 1 < j < i$$

The last decision made requires one to choose mn from {1, 2, 3,.....un}

Once a value of mn has been chosen, the remaining decisions must be such as to use the remaining funds C – Cn mn in an optimal way.

The principle of optimality holds on

$$f_n \sim C \sim \sim \max \{ On (m_n) fn \_ 1 (C - C_n m_n) \} \; 1 < m_n < u_n$$

for any fi (xi), i > 1, this equation generalizes to

$$f_n (x) = \max \{ ci (mi) fi - 1 (x - Ci mi) \} \; 1 < mi < ui$$

clearly, f0 (x) = 1 for all x, 0 < x < C and f (x) = -oo for all x < 0. Let

S^i consist of tuples of the form (f, x), where f = fi (x).

There is atmost one tuple for each different 'x', that result from a sequence of decisions on m1, m2, ... mn. The dominance rule (f1, x1) dominate (f2, x2) if f1 ≥ f2 and x1 ≤ x2. Hence, dominated tuples can be discarded from S^i.

Example 1:

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

Solution:

We assume that if if stage I has mi devices of type i in parallel, then ϕ i (mi) =1 – (1- ri)^{mi}

Since, we can assume each ci > 0, each mi must be in the range 1 ≤ mi ≤ ui. Where:

$$u_i = \sim \; | C + C_i^{\,n} - C \sum_{j=1} C_i \sim$$

Using the above equation compute u1, u2 and u3.

$$u1 = \frac{105 + 30 - (30+15+20)}{30} = \frac{70}{30} = 2$$

$$u2 = \frac{105+15-(30+15+20)}{15} = \frac{55}{15} = 3$$

$$u3 = \frac{105+20-(30+15+20)}{20} = \frac{60}{20} = 3$$

We use S -* *i:stage number and J*: *no. of devices in stage i* = mi S°

= $\{f_o(x), x\}$ *initially* $f_o(x) = 1$ *and* $x = 0$, *so,* $S^o = \{1, 0\}$

Compute S^1, S^2 and S^3 as follows:

S1 = depends on u1 value, as u1 = 2, so

$$S1 = \{S1, S^1\}$$
$$1 \quad 2$$

S2 = depends on u2 value, as u2 = 3, so

$$S2 = \{S^2, S^2, S^2\}$$
$$1 \quad 2 \quad 3$$

S3 = depends on u3 value, as u3 = 3, so

$$S3 = \{S^3, S^3, S^3\}$$
$$1 \quad 2 \quad 3$$

Now find $S^1(x), x$

$f1(x) = \{01(1) f_o \sim\sim, 01(2) f0()\}$ With devices m1 = 1 and m2 = 2 Compute Ø1 (1) and Ø1 (2) using the formula: $Øi(mi)) = 1 - (1 - ri)\, mi$

$\sim\sim 1 \sim\sim 1 \sim\sim 1 \sim r \sim m1 = 1 - (1 - 0.9)^1 = 0.9$

$1_1 \quad 1 \sim (2) = 1 - (1 - 0.9)\, 2 = 0.99$

$S \quad \sim\sim f1 \sim x \sim, x \sim\sim$

$1 \quad \sim \sim 0.9, 30$

$S2$

$1 = 10.99, 30 + 30\} = (0.99, 60$

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

Next find $S^2_1 \sim\sim 2f(x), x \sim\sim$

$f2(x) = \{02(1) * f1(), 02(2) * f1(), 02(3) * f1()\}$

~2 ~1~ ~ 1 ~~1 ~ *rI* ~ $\bar{\bar{m}}i$ 1 – (1 – 0.8) = 1 $_1$ 0.2 = 0.8

~$_2$~2~ ~ 1 ~~1 ~ 0.8~ 2 = 0.96

0_2(3) = 1 - (1 - 0.8) 3 = 0.992

= {(0.8(0.9),30 + 15), (0.8(0.99),60 + 15)} = {(0.72, 45), (0.792, 75)} =
{(0.96(0.9),30 + 15 +15) , (0.96(0.99),60 + 15 + 15)}
= {(0.864, 60), (0.9504, 90)}

= {(0.992(0.9),30 + 15 +15+15) , (0.992(0.99),60 + 15 + 15+15)}
= {(0.8928, 75), (0.98208, 105)}
$S2 = \{S^2_1, S^2_2, S^2_3\}$

By applying Dominance rule to S^2:

Therefore, S2 = {(0.72, 45), (0.864, 60), (0.8928, 75)} Dominance Rule:

If S^i contains two pairs (f1, x1) and (f2, x2) with the property that f1 ≥ f2 and x1 ≤ x2, then (f1, x1) dominates (f2, x2), hence by dominance rule (f2, x2) can be discarded. Discarding or pruning rules such as the one above is known as dominance rule. Dominating tuples will be present in S^i and Dominated tuples has to be discarded from Si.

Case 1: if f1 ≤ f2 and x1 > x2 then discard (f1, x1)

Case 2: if f1 > f2 and x1 < x2 the discard (f2, x2)

Case 3: otherwise simply write (f1, x1)

S2 = {(0.72, 45), (0.864, 60), (0.8928, 75)}

Ø 3 (1) = 1 ~ ~1 _ *rI* ~ *mi* = 1 – (1 – 0.5)^1 = 1 – 0.5 = 0.5

Ø $_2$ ~2~ ~ 1 ~ ~1 = 0.75
~ *S* 0.5~ 2
3
= 0.875
Ø S^2 ~3~ ~ 1 ~ ~1

~ 0.5~ 3
3
S_{3}
2

S 13 = {(0.5 (0.72), 45 + 20), (0.5 (0.864), 60 + 20), (0.5 (0.8928), 75 + 20)}

S 13 = {(0.36, 65), (0.437, 80), (0.4464, 95)}

S_2^3 ={(0.75 (0.72), 45 + 20 + 20), (0.75 (0.864), 60 + 20 +20),
(0.75 (0.8928), 75 + 20 + 20)}

= {(0.54, 85), (0.648, 100), (0.6696, 115)}

S_3^3 = { □0.875 (0.72), 45 + 20 + 20 + 20), □0.875 (0.864),60 + 20 + 20 + 20,
□0.875 (0.8928), 75 + 20 + 20 + 20□ }

S^3

3 = {(0.63, 105), (1.756, 120), (0.7812, 135)}

If cost exceeds 105, remove that tuples

S3 = {(0.36, 65), (0.437, 80), (0.54, 85), (0.648, 100)}

The best design has a reliability of 0.648 and a cost of 100. Tracing back for the solution through S^i 's we can determine that m3 = 2, m2 = 2 and m1 = 1.

Other Solution:

According to the principle of optimality:

fn(C) = max {~n (mn). fn-1 (C - Cn mn) with fo (x) = 1 and $0 \leq x \leq C$; 1 ~ $mn < un$

Since, we can assume each ci > 0, each mi must be in the range $1 \leq mi \leq ui$. Where:

S2 ={(0.75 (0.72), 45 + 20 + 20), (0.75 (0.864), 60 + 20 +20),

$$= \sim iC + Ci \_ \sim CJ \text{ r } / Ci \text{ I} \sim$$

Using the above equation compute u1, u2 and u3.

$$u1 = \frac{105 \square \square 30 \square +}{30} \sim \frac{70}{30} = 2$$

$$u2 = \frac{105 \square 15 \square 30 \square +}{15} \sim \frac{55}{15} \sim 3$$

$$u3 = \frac{105 \square 15 \square 30 \square + 20 \square}{20} = \frac{60}{20} = 3$$

f3 (105) = max {~3 (m3). f2 (105 - 20m3)} $1 < m3 ! u3$

= max {3(1) f2(105 - 20), 63(2) f2(105 - 20x2), ~3(3) f2(105 -20x3)} = max {0.5 f2(85), 0.75 f2(65), 0.875 f2(45)}

= max {0.5 x 0.8928, 0.75 x 0.864, 0.875 x 0.72} = 0.648.

= max {2 (m2). f1 (85 -15m2)}
$1 ! m2 ! u2$

f2 (85) = max {2(1).f1(85 - 15), ~2(2).f1(85 - 15x2), ~2(3).f1(85 - 15x3)} = max {0.8 f1(70), 0.96 f1(55), 0.992 f1(40)}

= max {0.8 x 0.99, 0.96 x 0.9, 0.99 x 0.9} = 0.8928

f1 (70) = max {~1(m1). f0(70 - 30m1)}

$1 ! m1 ! u1$

= max {~1(1) f0(70 - 30), t1(2) f0 (70 - 30x2)}

= max {~1(1) x 1, $_{t1(2)}$ x 1} = max {0.9, 0.99} = 0.99

f1 (55) = max {t1(m1). f0(55 - 30m1)}

1 ! *m*1 ! *u*1

= max {~1(1) f0(50 - 30), t1(2) f0(50 - 30x2)}

= max {~1(1) x 1, $_{t1(2)}$ x -oo} = max {0.9, -oo} = 0.9

f1 (40) = max {~1(m1). f0 (40 - 30m1)}

1 ! *m*1 ! *u*1

= max {~1(1) f0(40 - 30), t1(2) f0(40 - 30x2)}

= max {~1(1) x 1, $_{t1(2)}$ x -oo} = max{0.9, -oo} = 0.9

f2 (65) = max {2(m2). f1(65 -15m2)}

1 ! *m*2 ! *u*2

= max {2(1) f1(65 - 15), 62(2) f1(65 - 15x2), ~2(3) f1(65 - 15x3)} = max {0.8 f1(50), 0.96 f1(35), 0.992 f1(20)}

= max {0.8 x 0.9, 0.96 x 0.9, -oo} = 0.864

f1 (50) = max {~1(m1). f0(50 - 30m1)}

1 ! *m*1 ! *u*1

= max {~1(1) f0(50 - 30), t1(2) f0(50 - 30x2)}

= max {~1(1) x 1, $_{t1(2)}$ x -oo} = max{0.9, -oo} = 0.9 f1 (35) = max ~1(m1). f0(35 - 30m1)}

1 ! *m*1 ! *u*1

= max {~1(1).f0(35-30), ~1(2).f0(35-30x2)}

= max {~1(1) x 1, $_{t1(2)}$ x -oo} = max{0.9, -oo} = 0.9

f1 (20) = max {~1(m1). f0(20 - 30m1)}

1 ! *m*1 ! *u*1

= max {~1(1) f0(20 - 30), t1(2) f0(20 - 30x2)}

= max {~1(1) x -, ~1(2) x -oo} = max{-oo, -oo} = -oo

f2 (45) = max {2(m2). f1(45 -15m2)}

1 ! *m*2 ! *u*2

= max {2(1) f1(45 - 15), ~2(2) f1(45 - 15x2), ~2(3) f1(45 - 15x3)} = max {0.8 f1(30), 0.96 f1(15), 0.992 f1(0)}

= max {0.8 x 0.9, 0.96 x -, 0.99 x -oo} = 0.72

f1 (30) = max {~1(m1). f0(30 - 30m1)} 1<*m*1~*u*1

= max {~1(1) f0(30 - 30), t1(2) f0(30 - 30x2)}

= max {~1(1) x 1, $_{t1(2)}$ x -oo} = max{0.9, -oo} = 0.9 Similarly, f1 (15) = -,

f1 (0) = -.

The best design has a reliability = 0.648 and

Cost = 30 x 1 + 15 x 2 + 20 x 2 = 100.

Tracing back for the solution through S^i 's we can determine that: m3 = 2, m2 = 2 and

m1 = 1.

NRCM
your roots to success.