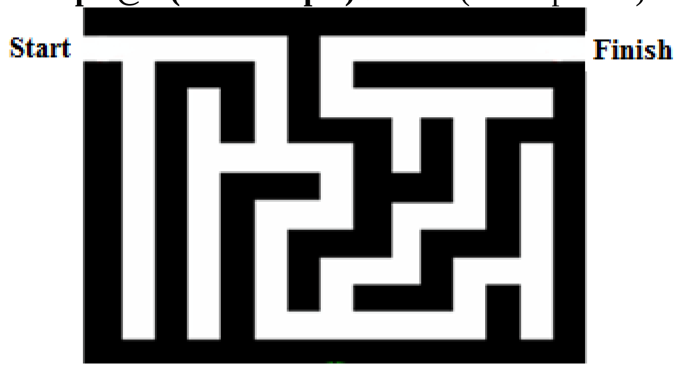
UNIT IV:
Backtracking: General method, applications-n-queen problem, sum of subsets problem, graph coloring, Hamiltonian cycles.
Branch and Bound: General method, applications - Travelling sales person problem,0/1 knapsack problem- LC Branch and Bound solution, FIFO Branch and Bound solution.

Backtracking (General method)

Many problems are difficult to solve algorithmically. Backtracking makes it possible to solve at least some large instances of difficult combinatorial problems.
Suppose you have to make a series of decisions among various choices, where

- You don't have enough information to know what to choose
- Each decision leads to a new set of choices.
- Some sequence of choices (more than one choices) may be a solution to your problem.

Backtracking is a methodical (Logical) way of trying out various sequences of decisions, until you find one that "works"
Example@1 (net example) : Maze (a tour puzzle)



Given a maze, find a path from start to finish.

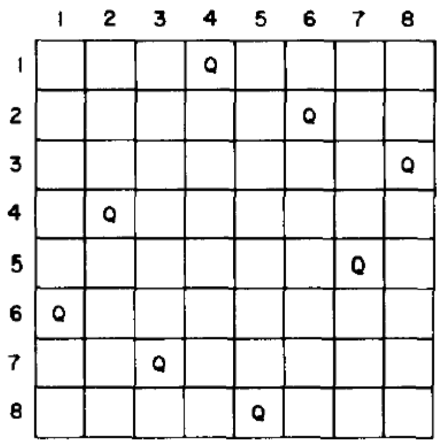
- In maze, at each intersection, you have to decide between 3 or fewer choices:
  - ✓ Go straight
  - ✓ Go left
  - ✓ Go right
- You don't have enough information to choose correctly
- Each choice leads to another set of choices.
- One or more sequences of choices may or may not lead to a solution.
- Many types of maze problem can be solved with backtracking.

Example@ 2 (text book):
Sorting the array of integers in a[1:n] is a problem whose solution is expressible by an n-tuple
$x_i \rightarrow$ is the index in 'a' of the i^{th} smallest element.
The criterion function 'P' is the inequality $a[x_i] \leq a[x_{i+1}]$ for $1 \leq i \leq n$
$S_i \rightarrow$ is finite and includes the integers 1 through n.
$m_i \rightarrow$size of set S_i
$m=m_1m_2m_3$---m_n n tuples that possible candidates for satisfying the function P.
With brute force approach would be to form all these n-tuples, evaluate (judge) each one with P and save those which yield the optimum.
By using backtrack algorithm; yield the same answer with far fewer than 'm' trails.
Many of the problems we solve using backtracking requires that all the solutions satisfy a complex set of constraints.
For any problem these constraints can be divided into two categories:

- ➢ Explicit constraints.
- ➢ Implicit constraints.

Explicit constraints: Explicit constraints are rules that restrict each $\mathbf{x_i}$ to take on values only from a given set.
Example: $\mathbf{x_i \geq 0}$ or si={all non negative real numbers}
X_i=0 or 1 or S_i={0, 1}
$l_i \leq x_i \leq u_i$ or s_i={a: $l_i \leq a \leq u_i$ }
The explicit constraint depends on the particular instance I of the problem being solved. All tuples that satisfy the explicit constraints define a possible solution space for I.
Implicit Constraints:
The implicit constraints are rules that determine which of the tuples in the solution space of I satisfy the criterion function. Thus implicit constraints describe the way in which the $\mathbf{X_i}$ must relate to each other.
<u>Applications of Backtracking:</u>
- ➢ N Queens Problem
- ➢ Sum of subsets problem
- ➢ Graph coloring
- ➢ Hamiltonian cycles.

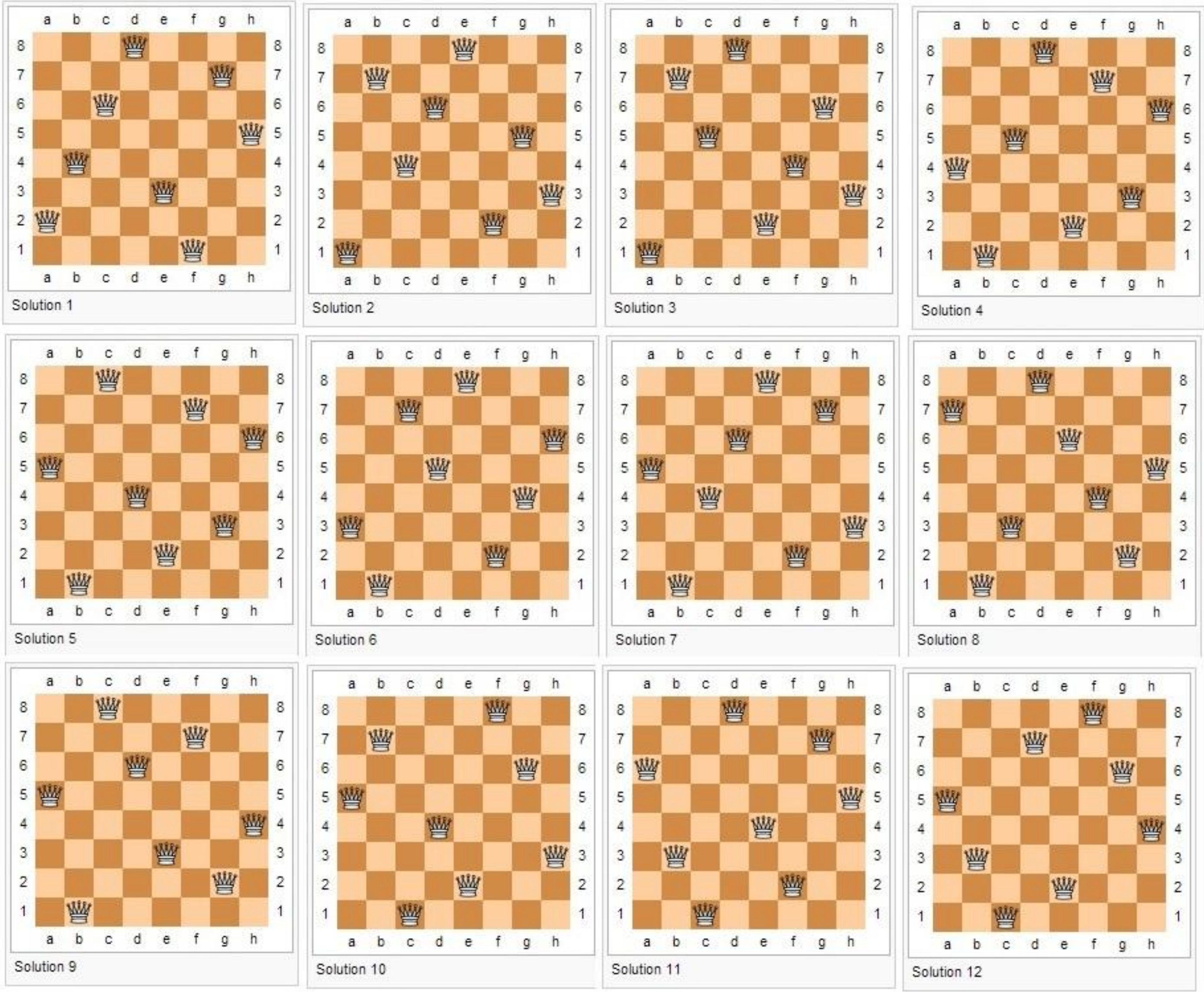
<u>N-Queens Problem:</u>
It is a classic combinatorial problem. The eight queen's puzzle is the problem of placing eight queens puzzle is the problem of placing eight queens on an 8×8 chessboard so that no two queens attack each other. That is so that no two of them are on the same row, column, or diagonal.
The 8-queens puzzle is an example of the more general n-queens problem of placing n queens on an n×n chessboard.



One solution to the 8-queens problem

Here queens can also be numbered 1 through 8
Each queen must be on a different row
Assume queen 'i' is to be placed on row 'i'
All solutions to the 8-queens problem can therefore be represented a s s-tuples(x_1, x_2, x_3—x_8)
xi→ the column on which queen 'i' is placed
si→{1, 2, 3, 4, 5, 6, 7, 8}, $1 \leq i \leq 8$
Therefore the solution space consists of 8^8 s-tuples.
The implicit constraints for this problem are that no two x_i's can be the same column and no two queens can be on the same diagonal.
By these two constraints the size of solution pace reduces from 88 tuples to 8! Tuples.
Form example s_i(4,6,8,2,7,1,3,5)

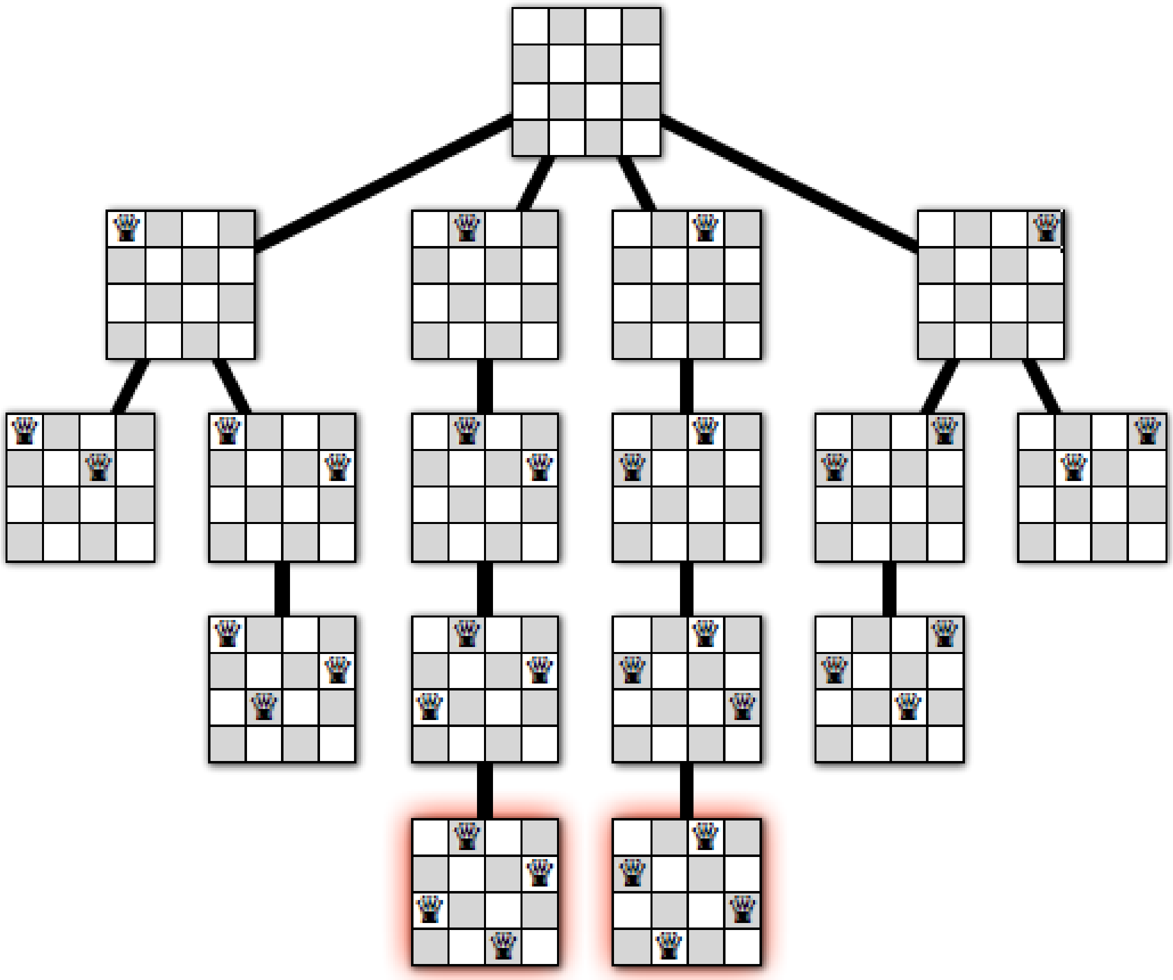
In the same way for n-queens are to be placed on an n×n chessboard, the solution space consists of all n! Permutations of n-tuples (1,2,----n).



Some solution to the 8-Queens problem

Algorithm for new queen be placed	**All solutions to the n·queens problem**

Algorithm Place(k,i)
//Return true if a queen can be placed in kth row & ith column
//Other wise return false
{
for j:=1 to k-1 do
if(x[j]=i or Abs(x[j]-i)=Abs(j-k)))
then return false
return true
}
	Algorithm NQueens(k, n)
	// its prints all possible placements of n-queens on an n×n chessboard.
	{
	for i:=1 to n do{
	if Place(k,i) then
	{
	X[k]:=I;
	if(k==n) then write (x[1:n]);
	else NQueens(k+1, n);
	}
	}}



The complete recursion tree for our algorithm for the 4 queens problem.

Sum of Subsets Problem:

Given positive numbers w_i $1 \leq i \leq n$, & m, here sum of subsets problem is finding all subsets of w_i whose sums are m.

Definition: Given n distinct +ve numbers (usually called weights), desire (want) to find all combinations of these numbers whose sums are m. this is called sum of subsets problem.

To formulate this problem by using either fixed sized tuples or variable sized tuples.

Backtracking solution uses the fixed size tuple strategy.

For example:

If n=4 (w_1, w_2, w_3, w_4)=(11,13,24,7) and m=31.

Then desired subsets are (11, 13, 7) & (24, 7).

The two solutions are described by the vectors (1, 2, 4) and (3, 4).

In general all solution are k-tuples (x_1, x_2, x_3---x_k) $1 \leq k \leq n$, different solutions may have different sized tuples.

- Explicit constraints requires $x_i \in \{j \,/\, j$ is an integer $1 \leq j \leq n \}$
- Implicit constraints requires:
  No two be the same & that the sum of the corresponding w_i's be m
  i.e., (1, 2, 4) & (1, 4, 2) represents the same. Another constraint is $x_i<x_{i+1}$ $1 \leq i \leq k$

$W_i \rightarrow$ weight of item i

M→ Capacity of bag (subset)
X_i→ the element of the solution vector is either one or zero.
X_i value depending on whether the weight wi is included or not.
If X_i=1 then wi is chosen.
If X_i=0 then wi is not chosen.

$$\underbrace{\sum_{i=1}^{k} W(i)X(i)}_{\text{Total sum till now}} + \underbrace{\sum_{i=k+1}^{n} W(i)}_{\text{Still there}} \ge M$$

The above equation specify that x_1, x_2, x_3, --- x_k cannot lead to an answer node if this condition is not satisfied.

$$\sum_{i=1}^{k} W(i)X(i) + W(k+1) > M$$

The equation cannot lead to solution.

$$B_k(X(1), \ldots, X(k)) = true \text{ iff } \left(\sum_{i=1}^{k} W(i)X(i) + \sum_{i=k+1}^{n} W(i) \ge M \text{ and } \sum_{i=1}^{k} W(i)X(i) + W(k+1) \le M \right)$$

$$s = \sum_{j=1}^{k-1} W(j)X(j). \quad \text{and} \quad r = \sum_{j=k}^{n} W(j)$$

Recursive backtracking algorithm for sum of subsets problem

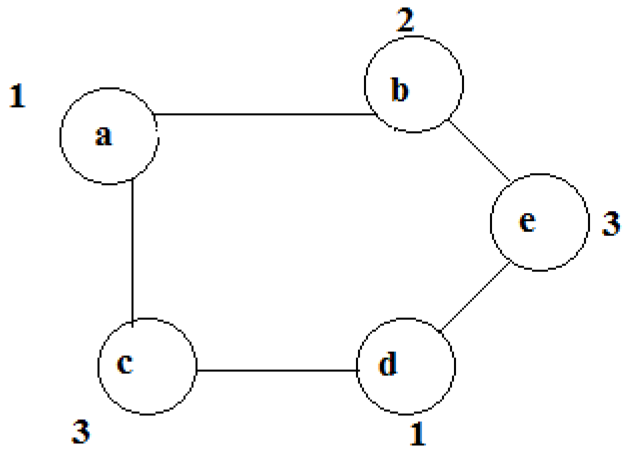
Algorithm SumOfSub(s, k, r)
{

$$// s = \sum_{j=1}^{k-1} W(j)X(j). \quad \text{and} \quad r = \sum_{j=k}^{n} W(j)$$

X[k]=1
If(S+w[k]=m) then write(x[1:]); // subset found.
Else if (S+w[k] + w{k+1] ≤ M)
Then SumOfSub(S+w[k], k+1, r-w[k]);
if ((S+r - w{k] ≥ M) and (S+w[k+1] ≤M)) then
{
X[k]=0;
SumOfSub(S, k+1, r-w[k]);
}
}

Graph Coloring:

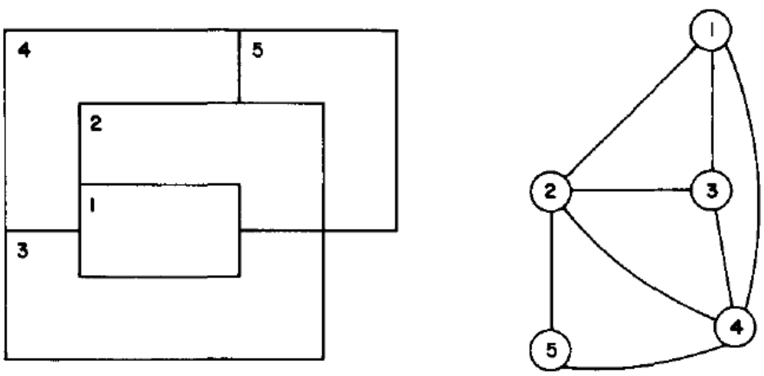
Let G be a undirected graph and 'm' be a given +ve integer. The graph coloring problem is assigning colors to the vertices of an undirected graph with the restriction that no two adjacent vertices are assigned the same color yet only 'm' colors are used.
The optimization version calls for coloring a graph using the minimum number of coloring.
The decision version, known as K-coloring asks whether a graph is colourable using at most k-colors.
Note that, if 'd' is the degree of the given graph then it can be colored with 'd+1' colors.
The m- colorability optimization problem asks for the smallest integer 'm' for which the graph G can be colored. This integer is referred as "**Chromatic number**" of the graph.
Example



- Above graph can be colored with 3 colors 1, 2, & 3.
- The color of each node is indicated next to it.
- 3-colors are needed to color this graph and hence this graph' Chromatic Number is 3.

- A graph is said to be planar iff it can be drawn in a plane (flat) in such a way that no two edges cross each other.
- **M-Colorability decision problem** is the 4-color problem for planar graphs.
- Given any map, can the regions be colored in such a way that no two adjacent regions have the same color yet only 4-colors are needed?
- To solve this problem, graphs are very useful, because a map can easily be transformed into a graph.
- Each region of the map becomes a node, and if two regions are adjacent, then the corresponding nodes are joined by an edge.

  - **Example:**



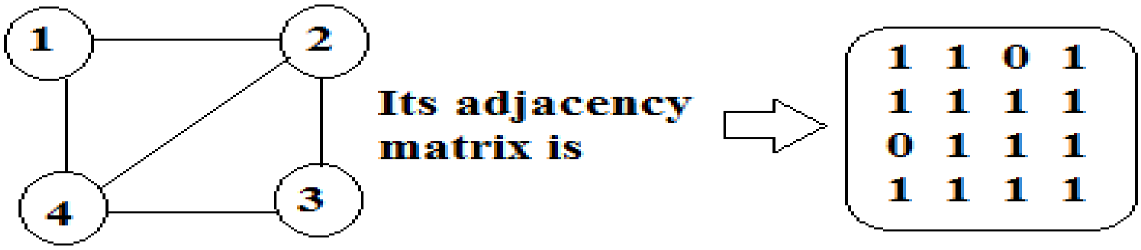
A map and its planar graph representation

The above map requires 4 colors.
- Many years, it was known that 5-colors were required to color this map.

- After several hundred years, this problem was solved by a group of mathematicians with the help of a computer. They show that 4-colors are sufficient.

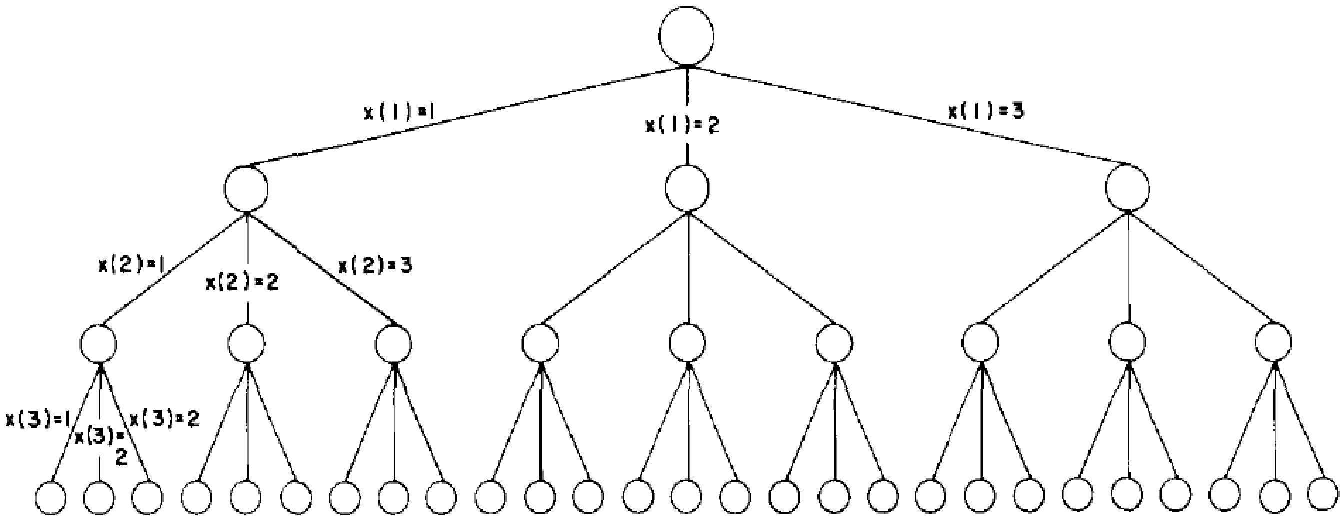
Suppose we represent a graph by its adjacency matrix G[1:n, 1:n]

Ex:



Here G[i, j]=1 if (i, j) is an edge of G, and G[i, j]=0 otherwise.
Colors are represented by the integers 1, 2,---m and the solutions are given by the n-tuple (x1, x2,---xn)
xi→ Color of node i.

State Space Tree for
n=3→ nodes
m=3→colors



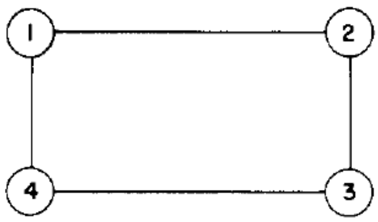
State space tree for MCOLORING when $n = 3$ and $m = 3$

1st node coloured in 3-ways
2nd node coloured in 3-ways
3rd node coloured in 3-ways
So we can colour in the graph in 27 possibilities of colouring.

Finding all m-coloring of a graph	Getting next color

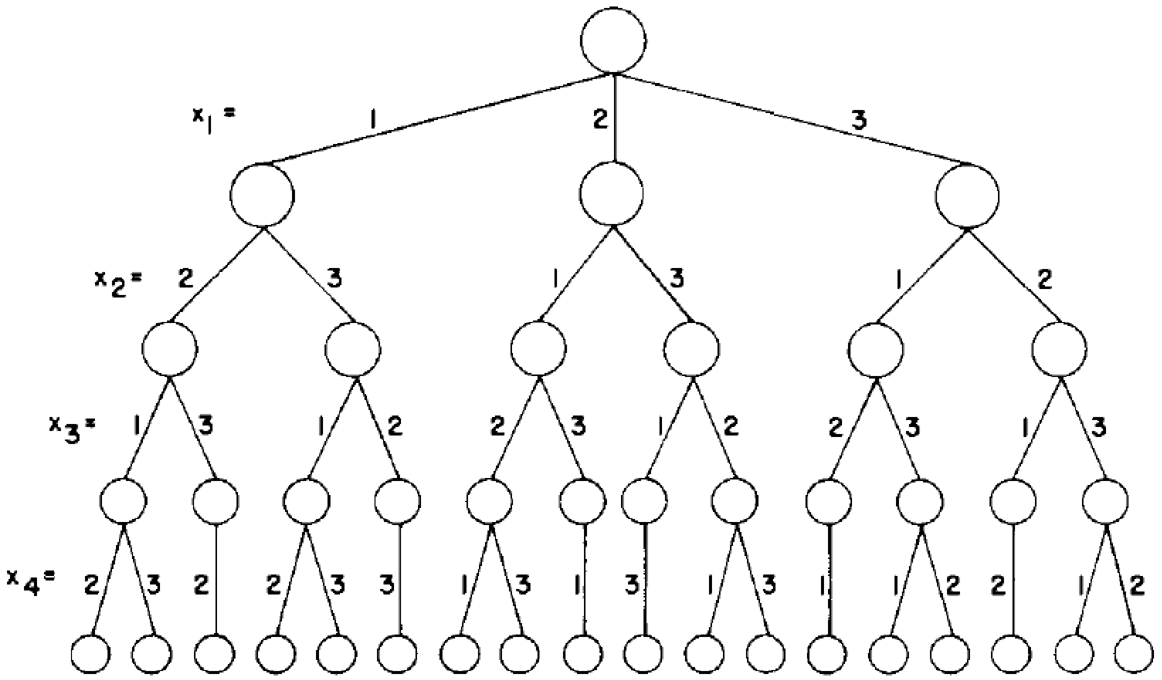
Algorithm mColoring(k){
// g(1:n, 1:n)→ boolean adjacency matrix.
// k→index (node) of the next vertex to color.
repeat{
nextvalue(k); // assign to x[k] a legal color.
if(x[k]=0) then return; // no new color possible
if(k=n) then write(x[1: n];
else mcoloring(k+1);
}
until(false)
}
	Algorithm NextValue(k){
	//x[1],x[2],---x[k-1] have been assigned integer values in the range [1, m]
	repeat {
	x[k]=(x[k]+1)mod (m+1); //next highest color
	if(x[k]=0) then return; // all colors have been used.
	for j=1 to n do
	{
	if ((g[k,j]≠0) and (x[k]=x[j]))
	then break;
	}
	if(j=n+1) then return; //new color found
	} until(false)
	}

Previous paper example:



Adjacency matrix is

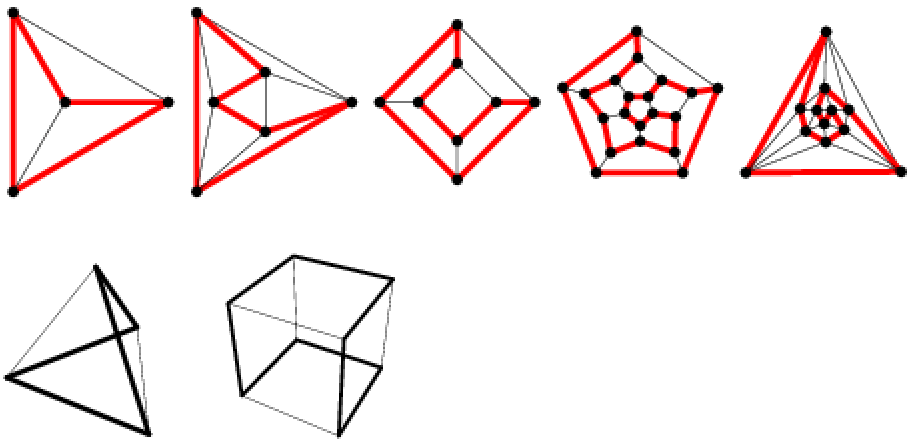
$$\begin{pmatrix} 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix}$$



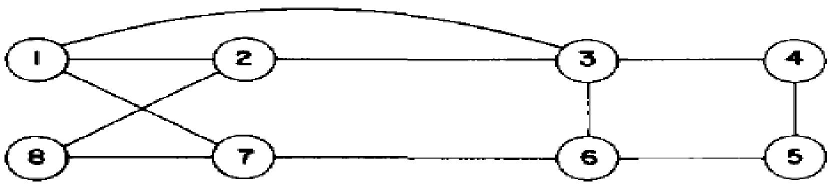
A 4 node graph and all possible 3 colorings

Hamiltonian Cycles:

- **Def:** Let G=(V, E) be a connected graph with n vertices. A Hamiltonian cycle is a round trip path along n-edges of G that visits every vertex once & returns to its starting position.
- It is also called the Hamiltonian circuit.
- Hamiltonian circuit is a graph cycle (i.e., closed loop) through a graph that visits each node exactly once.
- A graph possessing a Hamiltonian cycle is said to be Hamiltonian graph.
  Example:

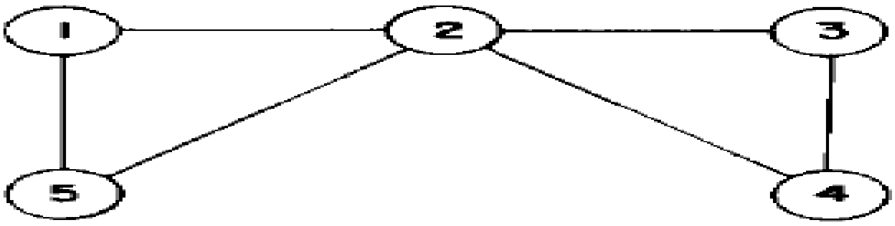


- In graph G, Hamiltonian cycle begins at some vertiex v1 $\in$ G and the vertices of G are visited in the order $v_1, v_2, --- v_{n+1}$, then the edges (v_i, v_{i+1}) are in E, $1 \le i \le n$.



→g1

The above graph contains Hamiltonian cycle: 1,2,8,7,6,5,4,3,1



The above graph contains no Hamiltonian cycles.

- There is no known easy way to determine whether a given graph contains a Hamiltonian cycle.
- By using backtracking method, it can be possible
  - Backtracking algorithm, that finds all the Hamiltonian cycles in a graph.
  - The graph may be directed or undirected. Only distinct cycles are output.
  - From graph g1 backtracking solution vector= {1, 2, 8, 7, 6, 5, 4, 3, 1}
  - The backtracking solution vector $(x_1, x_2, --- x_n)$
    x_i→ i^{th} visited vertex of proposed cycle.

- By using backtracking we need to determine how to compute the set of possible vertices for x_k if x_1, x_2, x_3---x_{k-1} have already been chosen.
  If k=1 then x1 can be any of the n-vertices.

By using "NextValue" algorithm the recursive backtracking scheme to find all Hamiltoman cycles.

This algorithm is started by 1^{st} initializing the adjacency matrix G[1:n, 1:n] then setting x[2:n] to zero & x[1] to 1, and then executing Hamiltonian (2)

Generating Next Vertex	**Finding all Hamiltonian Cycles**

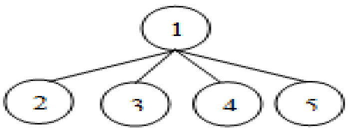
Algorithm NextValue(k)
{
// x[1: k-1]→ is path of k-1 distinct vertices.
// if x[k]=0, then no vertex has yet been assigned to x[k]
Repeat{
X[k]=(x[k]+1) mod (n+1); //Next vertex
If(x[k]=0) then return;
If(G[x[k-1], x[k]]≠0) then
{
For j:=1 to k-1 do if(x[j]=x[k]) then break;
//Check for distinctness
If(j=k) then //if true , then vertex is distinct
If((k<n) or (k=n) and G[x[n], x[1]]≠0))
Then return ;
}
}
Until (false);
}
	Algorithm Hamiltonian(k)
	{
	Repeat{
	NextValue(k); //assign a legal next value to x[k]
	If(x[k]=0) then return;
	If(k=n) then write(x[1:n]);
	Else Hamiltonian(k+1);
	} until(false)
	}

Branch & Bound

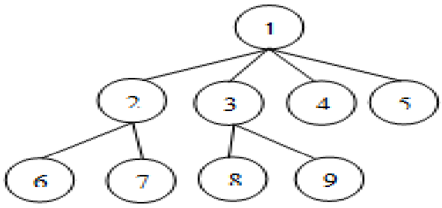
Branch & Bound (B & B) is general algorithm (or Systematic method) for finding optimal solution of various optimization problems, especially in discrete and combinatorial optimization.

- The B&B strategy is very similar to backtracking in that a state space tree is used to solve a problem.
- The differences are that the B&B method
  - ✓ Does not limit us to any particular way of traversing the tree.
  - ✓ It is used only for optimization problem
  - ✓ It is applicable to a wide variety of discrete combinatorial problem.
- B&B is rather general optimization technique that applies where the greedy method & dynamic programming fail.
- It is much slower, indeed (truly), it often (rapidly) leads to exponential time complexities in the worst case.
- The term B&B refers to all state space search methods in which all children of the "E-node" are generated before any other "live node" can become the "E-node"
  - ✓ **Live node→** is a node that has been generated but whose children have not yet been generated.
  - ✓ **E-node→**is a live node whose children are currently being explored.

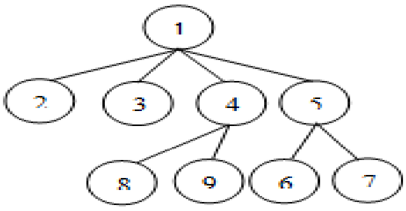
✓ **Dead node→** is a generated node that is not to be expanded or explored any further. All children of a dead node have already been expanded.



Live Node: 2, 3, 4, and 5



FIFO Branch & Bound (BFS)
Children of E-node are inserted in a queue.



LIFO Branch & Bound (D-Search)
Children of E-node are inserted in a stack.

➢ Two graph search strategies, BFS & D-search (DFS) in which the exploration of a new node cannot begin until the node currently being explored is fully explored.

➢ Both BFS & D-search (DFS) generalized to B&B strategies.

✓ **BFS→**like state space search will be called FIFO (First In First Out) search as the list of live nodes is "First-in-first-out" list (or queue).

✓ **D-search (DFS)→** Like state space search will be called LIFO (Last In First Out) search as the list of live nodes is a "last-in-first-out" list (or stack).

➢ In backtracking, bounding function are used to help avoid the generation of sub-trees that do not contain an answer node.

➢ We will use 3-types of search strategies in branch and bound

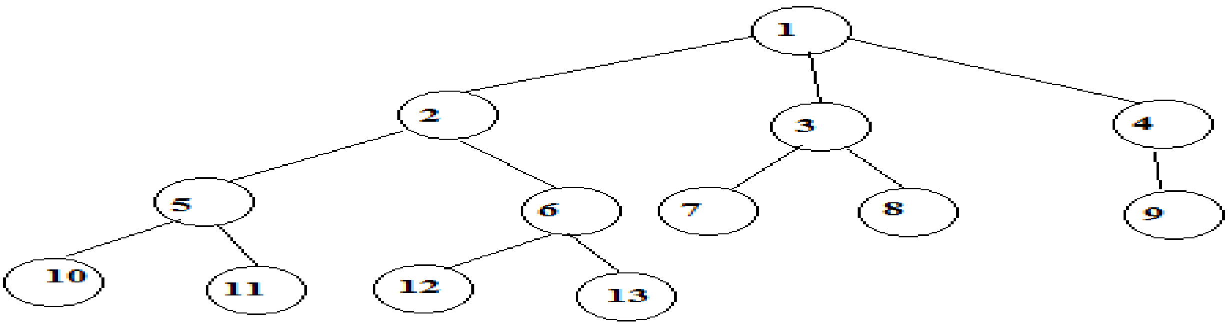
1) FIFO (First In First Out) search
2) LIFO (Last In First Out) search
3) LC (Least Count) search

FIFO B&B:

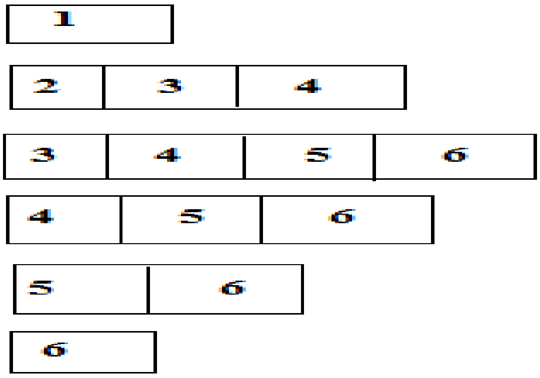
FIFO Branch & Bound is a BFS.
In this, children of E-Node (or Live nodes) are inserted in a queue.
Implementation of list of live nodes as a queue

✓ Least()→ Removes the head of the Queue

✓ Add()→ Adds the node to the end of the Queue

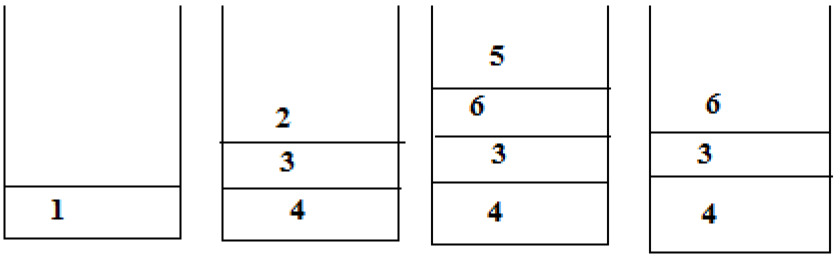


Assume that node '12' is an answer node in FIFO search, 1st we take E-node has '1'



LIFO B&B:
LIFO Brach & Bound is a D-search (or DFS).
In this children of E-node (live nodes) are inserted in a stack
Implementation of List of live nodes as a stack

- ✓ Least()→ Removes the top of the stack
- ✓ ADD()→Adds the node to the top of the stack.

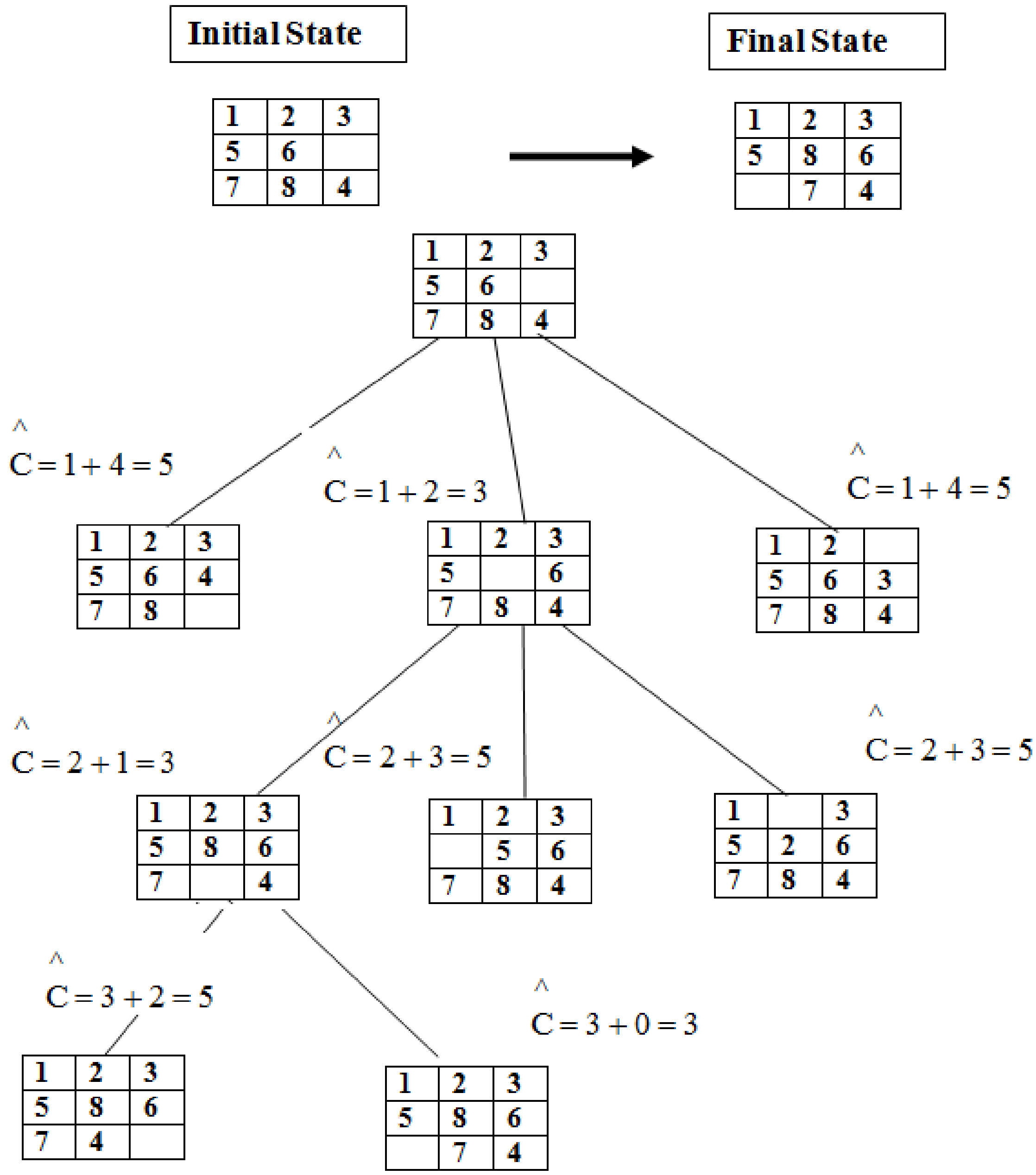


Least Cost (LC) Search:
The selection rule for the next E-node in FIFO or LIFO branch and bound is sometimes "blind". i.e., the selection rule does not give any preference to a node that has a very good chance of getting the search to an answer node quickly.

The search for an answer node can often be speeded by using an "intelligent" ranking function. It is also called an approximate cost function "**Ĉ**".
Expended node (E-node) is the live node with the best **Ĉ** value.
Branching: A set of solutions, which is represented by a node, can be partitioned into mutually (jointly or commonly) exclusive (special) sets. Each subset in the partition is represented by a child of the original node.
Lower bounding: An algorithm is available for calculating a lower bound on the cost of any solution in a given subset.

Each node X in the search tree is associated with a cost: **Ĉ(X)**
C=cost of reaching the current node, X(E-node) form the root + The cost of reaching an answer node form X.
Ĉ=g(X)+H(X).

Example:
8-puzzle
Cost function: **Ĉ** = g(x) +h(x)
where h(x) = the number of misplaced tiles
and g(x) = the number of moves so far
Assumption: move one tile in any direction cost 1.



Note: In case of tie, choose the leftmost node.

Travelling Salesman Problem:

Def:- Find a tour of minimum cost starting from a node S going through other nodes only once and returning to the starting point S.

Time Conmlexity of TSP for Dynamic Programming algorithm is $O(n^2 2^n)$

B&B algorithms for this problem, the worest case complexity will not be any better than $O(n^2 2^n)$ but good bunding functions will enables these B&B algorithms to solve some problem instances in much less time than required by the dynamic programming alogrithm.

Let $G=(V,E)$ be a directed graph defining an instances of TSP.

Let $C_{ij} \rightarrow$ cost of edge <i, j>

$C_{ij} = \infty$ if $<i, j> \notin E$

$|V|=n \rightarrow$ total number of vertices.

Assume that every tour starts & ends at vertex 1.

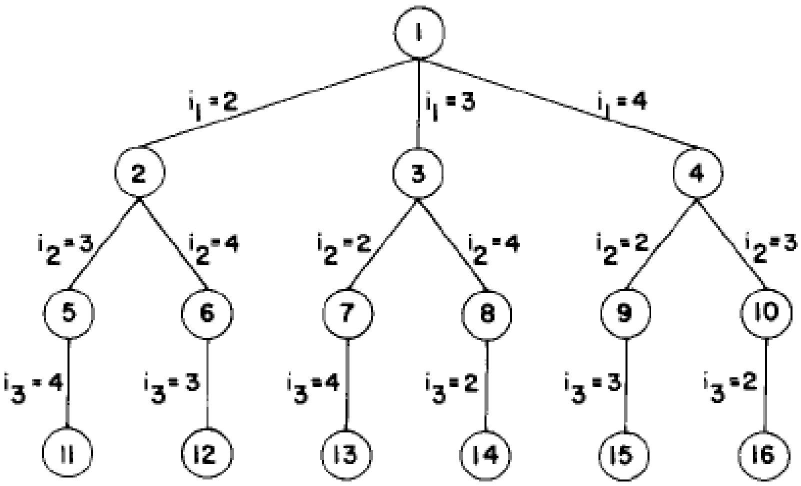
Solution Space S= {1, Π , 1 / Π is a permutation of (2, 3. 4. ----n) } then $|S|=(n-1)!$

The size of S reduced by restricting S

Sothat $(1, i_1, i2, \text{-----} i_{n-1}, 1) \in S$ iff $<i_j, i_{j+1}> \in E$. $O \le j \le n-1$, $i_0 - i_n = 1$

S can be organized into "State space tree".

Consider the following Example



State space tree for the travelling salesperson problem with n=4 and $i_0=i_4=1$

The above diagram shows tree organization of a complete graph with |V|=4.

Each leaf node 'L' is a solution node and represents the tour defined by the path from the root to L.

Node 12 represents the tour.

$i_0=1, i_1=2, i_2=4, i_3=3, i_4=1$

Node 14 represents the tour.

$i_0=1, i_1=3, i_2=4, i_3=2, i_4=1.$

<u>TSP is solved by using LC Branch & Bound:</u>

To use LCBB to search the travelling salesperson "State space tree" first define a cost function C(.) and other 2 functions $\hat{C}(.)$ & u(.)

Such that $\hat{C}(r) \le C(r) \le u(r)$ $\rightarrow$ for all nodes r.

Cost C(.)$\rightarrow$ is the solution node with least C(.) corresponds to a shortest tour in G.

C(A)={Length of tour defined by the path from root to A if A is leaf
Cost of a minimum-cost leaf in the sub-tree A, if A is not leaf }

From 1 $\hat{C}(r) \le C(r)$ then $\hat{C}(r)$ → is the length of the path defined at node A.

From previous example the path defined at node 6 is i_0, i_1, i_2=1, 2, 4 & it consists edge of <1,2> & <2,4>

Abetter $\hat{C}(r)$ can be obtained by using the reduced cost matrix corresponding to G.

- A row (column) is said to be reduced iff it contains at least one zero & remaining entries are non negative.
- A matrix is reduced iff every row & column is reduced.

$$\begin{bmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 18 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{bmatrix}$$

(a) Cost Matrix

$$\begin{bmatrix} \infty & 10 & 17 & 0 & 1 \\ 12 & \infty & 11 & 2 & 0 \\ 0 & 3 & \infty & 0 & 2 \\ 15 & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & 12 & \infty \end{bmatrix}$$

(b) Reduced Cost Matrix

L = 25

Given the following cost matrix:

$$\begin{bmatrix} inf & 20 & 30 & 10 & 11 \\ 15 & inf & 16 & 4 & 2 \\ 3 & 5 & inf & 2 & 4 \\ 19 & 6 & 18 & inf & 3 \\ 16 & 4 & 7 & 16 & inf \end{bmatrix}$$

- The TSP starts from node 1: <u>**Node 1**</u>
- Reduced Matrix: To get the lower bound of the path starting at node 1

Row # 1: reduce by 10 $\begin{bmatrix} inf & 10 & 20 \\ 15 & inf & 16 \\ 3 & 5 & inf & 2 \\ 19 & 6 & 18 & in \\ 16 & 4 & 7 & 16 \end{bmatrix}$	Row #2: reduce 2 $\begin{bmatrix} inf & 10 & 20 \\ 13 & inf & 14 \\ 3 & 5 & inf & 2 \\ 19 & 6 & 18 & in \\ 16 & 4 & 7 & 16 \end{bmatrix}$	Row #3: reduce by 2 $\begin{bmatrix} inf & 10 & 20 \\ 13 & inf & 14 \\ 1 & 3 & inf & 0 \\ 19 & 6 & 18 & in \\ 16 & 4 & 7 & 16 \end{bmatrix}$
Row # 4: Reduce by 3:	Row # 5: Reduce by 4	Column 1: Reduce by 1

$\begin{bmatrix} inf & 10 & 20 \\ 13 & inf & 14 \\ 1 & 3 & inf \\ 16 & 3 & 15 \\ 16 & 4 & 7 & 16 \end{bmatrix}$	$\begin{bmatrix} inf & 10 & 20 \\ 13 & inf & 14 \\ 1 & 3 & inf \\ 16 & 3 & 15 \\ 12 & 0 & 3 & 12 \end{bmatrix}$	$\begin{bmatrix} inf & 10 & 20 \\ 12 & inf & 14 \\ 0 & 3 & inf \\ 15 & 3 & 15 \\ 11 & 0 & 3 & 12 \end{bmatrix}$
Column 2: It is reduced.	Column 3: Reduce by 3 $\begin{bmatrix} inf & 10 & 17 & 0 & 1 \\ 12 & inf & 11 & 2 & 0 \\ 0 & 3 & inf & 0 & 2 \\ 15 & 3 & 12 & inf & 0 \\ 11 & 0 & 0 & 12 & inf \end{bmatrix}$	Column 4: It is reduced. Column 5: It is reduced.

The reduced cost is: RCL = 25
So the cost of node 1 is: Cost (1) = 25
The reduced matrix is:

Cost (1) = 25

$$\begin{bmatrix} inf & 10 & 17 & 0 & 1 \\ 12 & inf & 11 & 2 & 0 \\ 0 & 3 & inf & 0 & 2 \\ 15 & 3 & 12 & inf & 0 \\ 11 & 0 & 0 & 12 & inf \end{bmatrix}$$

- **Choose to go to vertex 2: Node 2**

- Cost of edge <1,2> is: A(1,2) = 10
- Set row #1 = inf since we are choosing edge <1,2>
- Set column # 2 = inf since we are choosing edge <1,2>
- Set A(2,1) = inf
- The resulting cost matrix is:

$$\begin{bmatrix} inf & inf & inf & inf & inf \\ inf & inf & 11 & 2 & 0 \\ 0 & inf & inf & 0 & 2 \\ 15 & inf & 12 & inf & 0 \\ 11 & inf & 0 & 12 & inf \end{bmatrix}$$

- The matrix is reduced:
- RCL = 0
- The cost of node 2 (Considering vertex 2 from vertex 1) is:

Cost(2) = cost(1) + A(1,2) = 25 + 10 = 35

- **<u>Choose to go to vertex 3: Node 3</u>**

- Cost of edge <1,3> is: A(1,3) = 17 (In the reduced matrix
- Set row #1 = inf since we are starting from node 1
- Set column # 3 = inf since we are choosing edge <1,3>
- Set A(3,1) = inf
- The resulting cost matrix is:

$$\begin{bmatrix} inf & inf & inf & inf & inf \\ 12 & inf & inf & 2 & 0 \\ inf & 3 & inf & 0 & 2 \\ 15 & 3 & inf & inf & 0 \\ 11 & 0 & inf & 12 & inf \end{bmatrix}$$

Reduce the matrix: Rows are reduced
The columns are reduced except for column # 1:
Reduce column 1 by 11:

$$\begin{bmatrix} inf & inf & inf & inf & inf \\ 1 & inf & inf & 2 & 0 \\ inf & 3 & inf & 0 & 2 \\ 4 & 3 & inf & inf & 0 \\ 0 & 0 & inf & 12 & inf \end{bmatrix}$$

The lower bound is: RCL = 11
The cost of going through node 3 is:
cost(3) = cost(1) + RCL + A(1,3) = 25 + 11 + 17 = 53

- <u>Choose to go to vertex 4: **Node 4**</u>

Remember that the cost matrix is the one that was reduced at the starting vertex 1
Cost of edge <1,4> is: A(1,4) = 0
Set row #1 = inf since we are starting from node 1
Set column # 4 = inf since we are choosing edge <1,4>
Set A(4,1) = inf
The resulting cost matrix is:

$$\begin{bmatrix} inf & inf & inf & inf & inf \\ 12 & inf & 11 & inf & 0 \\ 0 & 3 & inf & inf & 2 \\ inf & 3 & 12 & inf & 0 \\ 11 & 0 & 0 & inf & inf \end{bmatrix}$$

Reduce the matrix: Rows are reduced

Columns are reduced
The lower bound is: RCL = 0
The cost of going through node 4 is:
cost(4) = cost(1) + RCL + A(1,4) = 25 + 0 + 0 = 25

- **Choose to go to vertex 5: Node 5**

- Remember that the cost matrix is the one that was reduced at starting vertex 1
- Cost of edge <1,5> is: A(1,5) = 1
- Set row #1 = inf since we are starting from node 1
- Set column # 5 = inf since we are choosing edge <1,5>
- Set A(5,1) = inf
- The resulting cost matrix is:

$$\begin{bmatrix} inf & inf & inf & inf & inf \\ 12 & inf & 11 & 2 & inf \\ 0 & 3 & inf & 0 & inf \\ 15 & 3 & 12 & inf & inf \\ inf & 0 & 0 & 12 & inf \end{bmatrix}$$

Reduce the matrix:
Reduce rows:
Reduce row #2: Reduce by 2

$$\begin{bmatrix} inf & inf & inf & inf & inf \\ 10 & inf & 9 & 0 & inf \\ 0 & 3 & inf & 0 & inf \\ 15 & 3 & 12 & inf & inf \\ inf & 0 & 0 & 12 & inf \end{bmatrix}$$

Reduce row #4: Reduce by 3

$$\begin{bmatrix} inf & inf & inf & inf & inf \\ 10 & inf & 9 & 0 & inf \\ 0 & 3 & inf & 0 & inf \\ 12 & 0 & 9 & inf & inf \\ inf & 0 & 0 & 12 & inf \end{bmatrix}$$

Columns are reduced
The lower bound is: RCL = 2 + 3 = 5
The cost of going through node 5 is:
cost(5) = cost(1) + RCL + A(1,5) = 25 + 5 + 1 = 31

In summary:

So the live nodes we have so far are:

- ✓ 2: cost(2) = 35, path: 1->2
- ✓ 3: cost(3) = 53, path: 1->3
- ✓ 4: cost(4) = 25, path: 1->4
- ✓ 5: cost(5) = 31, path: 1->5

Explore the node with the lowest cost: Node 4 has a cost of 25

Vertices to be explored from node 4: 2, 3, and 5

Now we are starting from the cost matrix at node 4 is:

Cost (4) = 25

$$\begin{bmatrix} inf & inf & inf & inf & inf \\ 12 & inf & 11 & inf & 0 \\ 0 & 3 & inf & inf & 2 \\ inf & 3 & 12 & inf & 0 \\ 11 & 0 & 0 & inf & inf \end{bmatrix}$$

- ➢ **<u>Choose to go to vertex 2: Node 6 (path is 1->4->2)</u>**

Cost of edge <4,2> is: A(4,2) = 3

Set row #4 = inf since we are considering edge <4,2>

Set column # 2 = inf since we are considering edge <4,2>

Set A(2,1) = inf

The resulting cost matrix is:

$$\begin{bmatrix} inf & inf & inf & inf & inf \\ inf & inf & 11 & inf & 0 \\ 0 & inf & inf & inf & 2 \\ inf & inf & inf & inf & inf \\ 11 & inf & 0 & inf & inf \end{bmatrix}$$

Reduce the matrix: Rows are reduced

Columns are reduced

The lower bound is: RCL = 0

The cost of going through node 2 is:

cost(6) = cost(4) + RCL + A(4,2) = 25 + 0 + 3 = 28

➢ **<u>Choose to go to vertex 3: Node 7 (</u> path is 1->4->3)**

Cost of edge <4,3> is: A(4,3) = 12
Set row #4 = inf since we are considering edge <4,3>
Set column # 3 = inf since we are considering edge <4,3>
Set A(3,1) = inf
The resulting cost matrix is:

$$\begin{bmatrix} inf & inf & inf & inf & inf \\ 12 & inf & inf & inf & 0 \\ inf & 3 & inf & inf & 2 \\ inf & inf & inf & inf & inf \\ 11 & 0 & inf & inf & inf \end{bmatrix}$$

Reduce the matrix:

Reduce row #3: by 2:

$$\begin{bmatrix} inf & inf & inf & inf & inf \\ 12 & inf & inf & inf & 0 \\ inf & 1 & inf & inf & 0 \\ inf & inf & inf & inf & inf \\ 11 & 0 & inf & inf & inf \end{bmatrix}$$

Reduce column # 1: by 11

$$\begin{bmatrix} inf & inf & inf & inf & inf \\ 1 & inf & inf & inf & 0 \\ inf & 1 & inf & inf & 0 \\ inf & inf & inf & inf & inf \\ 0 & 0 & inf & inf & inf \end{bmatrix}$$

The lower bound is: RCL = 13
So the RCL of node 7 (Considering vertex 3 from vertex 4) is:
Cost(7) = cost(4) + RCL + A(4,3) = 25 + 13 + 12 = 50

- <u>Choose to go to vertex 5: **Node 8** (</u> path is 1->4->5)

Cost of edge <4,5> is: A(4,5) = 0
Set row #4 = inf since we are considering edge <4,5>
Set column # 5 = inf since we are considering edge <4,5>
Set A(5,1) = inf
The resulting cost matrix is:

$$\begin{bmatrix} inf & inf & inf & inf & inf \\ 12 & inf & 11 & inf & inf \\ 0 & 3 & inf & inf & inf \\ inf & inf & inf & inf & inf \\ inf & 0 & 0 & inf & inf \end{bmatrix}$$

Reduce the matrix:
Reduced row 2: by 11

$$\begin{bmatrix} inf & inf & inf & inf & inf \\ 1 & inf & 0 & inf & inf \\ 0 & 3 & inf & inf & inf \\ inf & inf & inf & inf & inf \\ inf & 0 & 0 & inf & inf \end{bmatrix}$$

Columns are reduced
The lower bound is: RCL = 11
So the cost of node 8 (Considering vertex 5 from vertex 4) is:
Cost(8) = cost(4) + RCL + A(4,5) = 25 + 11 + 0 = 36

In summary: So the live nodes we have so far are:

- ✓ 2: cost(2) = 35, path: 1->2
- ✓ 3: cost(3) = 53, path: 1->3
- ✓ 5: cost(5) = 31, path: 1->5
- ✓ 6: cost(6) = 28, path: 1->4->2
- ✓ 7: cost(7) = 50, path: 1->4->3
- ✓ 8: cost(8) = 36, path: 1->4->5

- Explore the node with the lowest cost: Node 6 has a cost of 28
- Vertices to be explored from node 6: 3 and 5
- Now we are starting from the cost matrix at node 6 is:

Cost (6) = 28

$$\begin{bmatrix} inf & inf & inf & inf & inf \\ inf & inf & 11 & inf & 0 \\ 0 & inf & inf & inf & 2 \\ inf & inf & inf & inf & inf \\ 11 & inf & 0 & inf & inf \end{bmatrix}$$

➢ **<u>Choose to go to vertex 3: Node 9 (</u> path is 1->4->2->3)**

Cost of edge <2,3> is: A(2,3) = 11
Set row #2 = inf since we are considering edge <2,3>
Set column # 3 = inf since we are considering edge <2,3>
Set A(3,1) = inf
The resulting cost matrix is:

$$\begin{bmatrix} inf & inf & inf & inf & inf \\ inf & inf & inf & inf & inf \\ inf & inf & inf & inf & 2 \\ inf & inf & inf & inf & inf \\ 11 & inf & inf & inf & inf \end{bmatrix}$$

Reduce the matrix: Reduce row #3: by 2

$$\begin{bmatrix} inf & inf & inf & inf & inf \\ inf & inf & inf & inf & inf \\ inf & inf & inf & inf & 0 \\ inf & inf & inf & inf & inf \\ 11 & inf & inf & inf & inf \end{bmatrix}$$

Reduce column # 1: by 11

$$\begin{bmatrix} inf & inf & inf & inf & inf \\ inf & inf & inf & inf & inf \\ inf & inf & inf & inf & 0 \\ inf & inf & inf & inf & inf \\ 0 & inf & inf & inf & inf \end{bmatrix}$$

The lower bound is: RCL = 2 +11 = 13
So the cost of node 9 (Considering vertex 3 from vertex 2) is:
Cost(9) = cost(6) + RCL + A(2,3) = 28 + 13 + 11 = 52

➢ **<u>Choose to go to vertex 5: Node 10 (</u> path is 1->4->2->5)**

Cost of edge <2,5> is: A(2,5) = 0
Set row #2 = inf since we are considering edge <2,3>
Set column # 3 = inf since we are considering edge <2,3>
Set A(5,1) = inf
The resulting cost matrix is:

$$\begin{bmatrix} inf & inf & inf & inf & inf \\ inf & inf & inf & inf & inf \\ 0 & inf & inf & inf & inf \\ inf & inf & inf & inf & inf \\ inf & inf & 0 & inf & inf \end{bmatrix}$$

Reduce the matrix: Rows reduced
Columns reduced
The lower bound is: RCL = 0
So the cost of node 10 (Considering vertex 5 from vertex 2) is:
Cost(10) = cost(6) + RCL + A(2,3) = 28 + 0 + 0 = 28

In summary: **So the live nodes we have so far are:**

- ✓ 2: cost(2) = 35, path: 1->2
- ✓ 3: cost(3) = 53, path: 1->3
- ✓ 5: cost(5) = 31, path: 1->5
- ✓ 7: cost(7) = 50, path: 1->4->3
- ✓ 8: cost(8) = 36, path: 1->4->5
- ✓ 9: cost(9) = 52, path: 1->4->2->3
- ✓ 10: cost(2) = 28, path: 1->4->2->5
- ➢ Explore the node with the lowest cost: Node 10 has a cost of 28
- ➢ Vertices to be explored from node 10: 3
- ➢ Now we are starting from the cost matrix at node 10 is:

Cost (10)=28

$$\begin{bmatrix} inf & inf & inf & inf & inf \\ inf & inf & inf & inf & inf \\ 0 & inf & inf & inf & inf \\ inf & inf & inf & inf & inf \\ inf & inf & 0 & inf & inf \end{bmatrix}$$

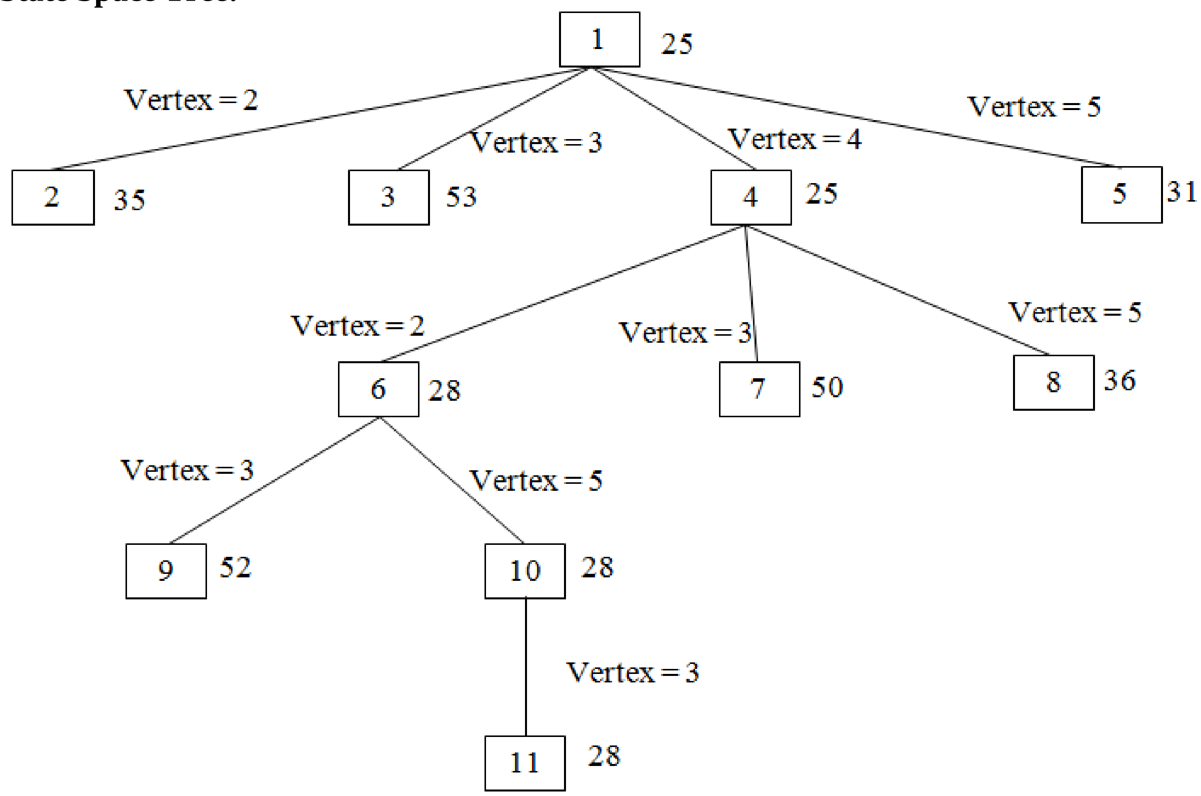
➢ **<u>Choose to go to vertex 3: Node 11 (</u> path is 1->4->2->5->3)**

Cost of edge <5,3> is: A(5,3) = 0
Set row #5 = inf since we are considering edge <5,3>
Set column # 3 = inf since we are considering edge <5,3>
Set A(3,1) = inf
The resulting cost matrix is:

$$\begin{bmatrix} inf & inf & inf & inf & inf \\ inf & inf & inf & inf & inf \\ inf & inf & inf & inf & inf \\ inf & inf & inf & inf & inf \\ inf & inf & inf & inf & inf \end{bmatrix}$$

Reduce the matrix: Rows reduced
Columns reduced
The lower bound is: RCL = 0
So the cost of node 11 (Considering vertex 5 from vertex 3) is:
Cost(11) = cost(10) + RCL + A(5,3) = 28 + 0 + 0 = 28

State Space Tree:



O/1 Knapsack Problem

What is Knapsack Problem: Knapsack problem is a problem in combinatorial optimization, Given a set of items, each with a mass & 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 & the total value is as large as possible.

O-1 Knapsack Problem can formulate as. Let there be n items, $\mathbf{Z_1}$ **to** $\mathbf{Z_n}$ where $\mathbf{Z_i}$ has value $\mathbf{P_i}$ & weight $\mathbf{w_i}$. The maximum weight that can carry in the bag is m.

All values and weights are non negative.

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

The formula can be stated as

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

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

$\mathbf{X_i=0 \text{ or } 1 \;\; 1 \le i \le n}$

To solve o/1 knapsack problem using B&B:

- Knapsack is a maximization problem
- Replace the objective function $\sum p_i x_i$ by the function $-\sum p_i x_i$ to make it into a minimization problem
- The modified knapsack problem is stated as

$$\text{Minimize } -\sum_{i=1}^{n} p_i x_i$$

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

$$x_i \in \{0,1\}, 1 \le i \le n$$

- Fixed tuple size solution space:
  - Every leaf node in state space tree represents an answer for which $\sum_{1 \le i \le n} w_i x_i \leqslant m$ is an answer node; other leaf nodes are infeasible
  - For optimal solution, define

$$c(x) = -\sum_{1 \le i \le n} p_i x_i$$

for every answer node x

- For infeasible leaf nodes, $c(x) = \infty$
- For non leaf nodes

$$c(x) = \min\{c(lchild(x)), c(rchild(x))\}$$

- Define two functions $\hat{c}(x)$ and $u(x)$ such that for every node x,

$$\hat{c}(x) \le c(x) \le u(x)$$

- Computing $\hat{c}(\cdot)$ and $u(\cdot)$

Let x be a node at level j, $1 \leq j \leq n+1$

Cost of assignment: $-\sum_{1 \leq i < j} p_i x_i$

$c(x) \leq -\sum_{1 \leq i < j} p_i x_i$

We can use $u(x) = -\sum_{1 \leq i < j} p_i x_i$

Using $q = -\sum_{1 \leq i < j} p_i x_i$, an improved upper bound function $u(x)$ is

$$u(x) = \text{ubound}(q, \sum_{1 \leq i < j} w_i x_i, j-1, m)$$

```
Algorithm  ubound ( cp, cw, k, m )
{
// Input:   cp: Current profit total
// Input:   cw: Current weight total
// Input:   k:  Index of last removed item
// Input:   m:  Knapsack capacity
b=cp; c=cw;
for i:=k+1 to n do{
      if(c+w[i] ≤ m) then {
              c:=c+w[i]; b=b-p[i];
      }
}
return b;
}
```

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 $k \geq 2$ disjoint sets V_i, $1 \leq i \leq k$. In addition, if <u, v> is an edge in E, then $u \in V_i$ and $v \in V_{i+1}$ for some i, $1 \leq i < k$.

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 V_i 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}, $1 \leq i \leq k - 2$, 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:

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

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:

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

Algorithm Bgraph (G, k, n, p)

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

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,

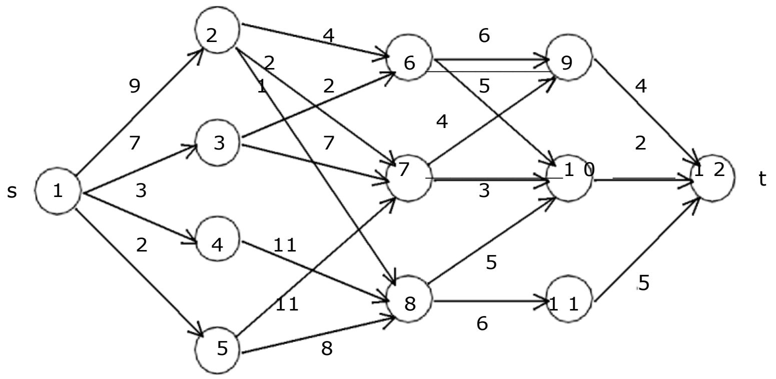
cost (2, 2) = min{c (2, 6) + cost (3, 6), c (2, 7) + cost (3, 7), c (2, 8) + cost (3, 8)} = min {4 + cost (3, 6), 2 + cost (3, 7), 1 + cost (3, 8)}

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

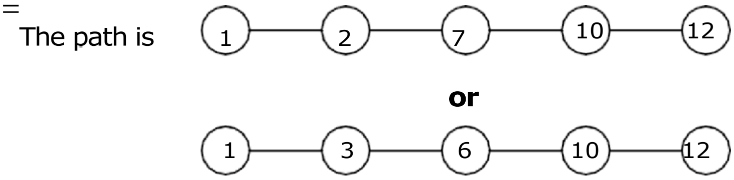
Therefore, cost (3, 6) = min {6 + 4, 5 + 2} = 7



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) =



min

{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

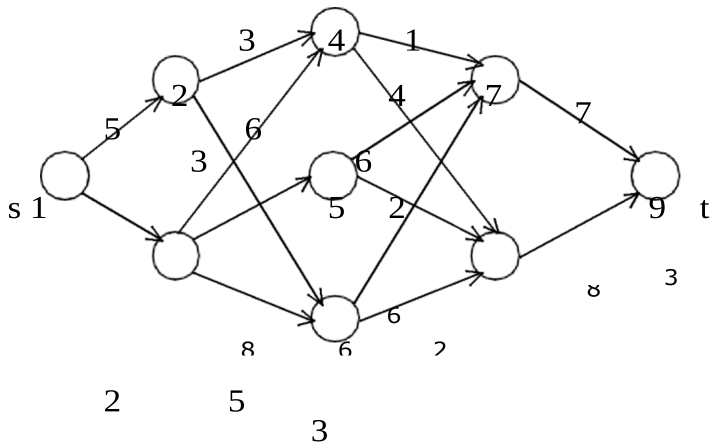
Bcost (3, 8) = min {9 + 1, 3 + 11, 2 + 8} = min {10, 14, 10} = 10 Bcost (4, 10) = min {9 + 5, 11 + 3, 10 + 5} = min {14, 14, 15) = 14

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:

$$\text{Bcost}(i, J) = \min_{\substack{l \in v_{i-1} \\ \langle l, j \rangle \in E}} \{\text{Bcost}(i-1, l) = c(l, J)\}$$

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:

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

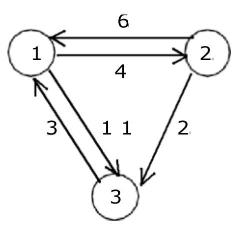
Algorithm All Paths (Cost, A, n)
// cost [1:n, 1:n] is the cost adjacency matrix of a graph which
// n vertices; A [I, j] is the cost of a shortest path from vertex
// i to vertex j. cost [i, i] = 0.0, for 1 ≤ i ≤ n.

```
{
        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) = $\begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \infty & 0 \end{bmatrix}$

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

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

$$A_{(1)} = \begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}$$

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

A2 (1, 1) = min {(A^1 (1, 2) + A^1 (2, 1), c (1, 1)} = min {(4 + 6), 0} + A^1 = 0
A2 (1, 2) = min {(A^1 (1, 2) (2, 2), c (1, 2)} = min {(4 + 0), 4} + A^1 (2, = 4
A2 (1, 3) = min {(A^1 (1, 2) 3), c (1, 3)} = min {(4 + 2), 11} = 6
A2 (2, 1) = min {(A (2, 2) + A (2, 1), c (2, 1)} = min {(0 + 6), 6} = 6
A2 (2, 2) = min {(A (2, 2) + A (2, 2), c (2, 2)} = min {(0 + 0), 0} = 0
A2 (2, 3) = min {(A (2, 2) + A (2, 3), c (2, 3)} = min {(0 + 2), 2} = 2
A2 (3, 1) = min {(A (3, 2) + A (2, 1), c (3, 1)} = min {(7 + 6), 3} = 3
A2 (3, 2) = min {(A (3, 2) + A (2, 2), c (3, 2)} = min {(7 + 0), 7} = 7
A2 (3, 3) = min {(A (3, 2) + A (2, 3), c (3, 3)} = min {(7 + 2), 0} = 0

$$A_{(2)} = \begin{bmatrix} 0 & 4 & 6 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}$$

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

107

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 ~ k ~ n ~c1k ~ g ~ k, V ~ ~ 1, k ~~ -- 1

~

min

Generalizing equation 1, we obtain (for i o S) -- 2

g (i, S) = min{ci j

j ES

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~ ~n-2~ .

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

~n-2~

~ ~n ~1~ ~ ~ k ~

k ~ 0 ~ ~

To calculate this sum, we use the binominal theorem:

[((n - 2) ((n - 2) ((n - 2) ((n - 2)1

n - 1 (n–1) 111 11+ii iI+ ii iI+----~~~ ~

~~ 0) ~ 1) ~ 2) ~(n~2)~~

According to the binominal theorem:

[((n - 2) ((n - 2) ((n - 2 ((n - 2)1

1 11+ii iI+ii ~~~~~~~~~~ ~~~=2n-2

~~0 ~ ~ 1 ~ ~ 2 ~ ~(n - 2))]

Therefore,

n - 1 ~ n _ 2'

~ (n _1~ ~ ~ k = (n - 1) 2n ~ 2

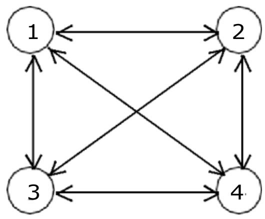
~

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

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 ≤ i ≤ 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 $\leq$ i $\leq$ 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:

$$\sum_{i=1}^{n} P_i \, (1 + d_i) \, .$$

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:

$$P(i) * level(a_i) \, .$$

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:

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

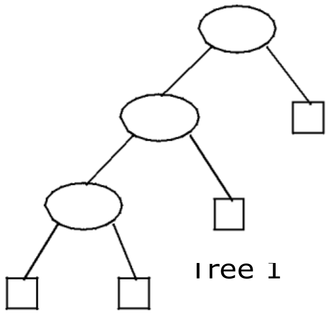
Given a fixed set of identifiers, we wish to create a binary search tree organization. We may expect different binary search trees for the same identifier set to have different performance characteristics.

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:

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

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:



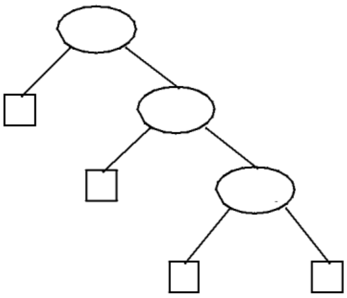
Tree 1

stop

if

do

Tree 2



do

if

stop

Tree 3

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

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

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

$$= \frac{1+2+3}{7} + \frac{1+2+3+3}{7} = \frac{6+9}{7} = \frac{15}{7}$$

$$\text{Cost (tree \# 4)} = \left(\frac{1}{7}\times 1 + \frac{1}{7}\times 2 + \frac{1}{7}\times 3\right) + \left(\frac{1}{7}\times 1 + \frac{1}{7}\times 2 + \frac{1}{7}\times 3 + \frac{1}{7}\times 3\right)$$

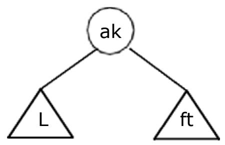
$$= \frac{1+2+3}{7} + \frac{1+2+3+3}{7} = \frac{6+9}{7} = \frac{15}{7}$$

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

$$= \frac{1+2+2}{7} + \frac{2+2+2+2}{7} = \frac{5+8}{7} = \frac{13}{7}$$

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:

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

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

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 $0 \leq i \leq n$.

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, $0 \leq i < 4$.

Solving for C (0, n):

First, computing all C (i, j) such that j - i = 1; j = i + 1 and as 0 $\leq$ 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 $\leq$ 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, 2) + C (3, 3)]}
= W (1, 3) + 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 $\leq$ 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 + 12), (8 + 3), (19 + 0)} = 14 + 11 = 25
ft (0, 3) = 2

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 + 9 = 11 = W 2) + C (3, 4)],

C (1, 4) (1, 4) + min {[C (1, 1) + C (2, 4)], [C (1,

[C (1, 3) + C (4, 4)]} + 8 = 19

= 11 + min {(0 + 8), (7 + 3), (12 + 0)} = 11 = 2

ft (1, 4)

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

Start with i = 0; so j = 4; as i < k $\leq$ 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.

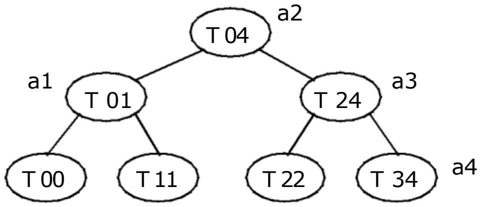
The left and right sub trees for 'T01' are 'T00' and 'T11' respectively. The root of T01 is 'a1'

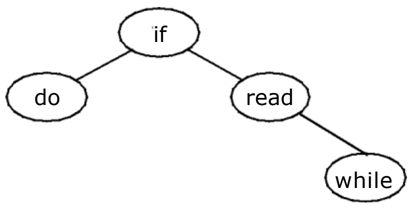
The left and right sub trees for T24 are T22 and T34 respectively.

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 $\leq$ i < 4; i = 0, 1, 2 and 3; i < k $\leq$ J. Start with i = 0; so j = 1; as i < k $\leq$ 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

C (1, 4) = W (1, 4) + min {[C (1, 1) + C (2, 4)], [C (1, 2) + C (3, 4)], [C (1, 3) + C (4, 4)]}

= 10 + min {(0 + 8), (6 + 3), (11 + 0)} = 10 + 8 = 18

ft (1, 4) = 2

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.

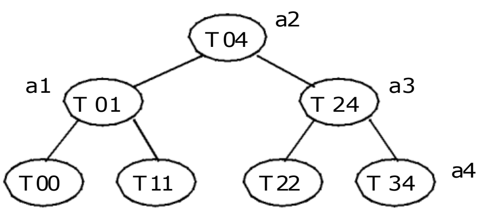
The left and right sub trees for 'T01' are 'T00' and 'T11' respectively. The root of T01 is 'a1'

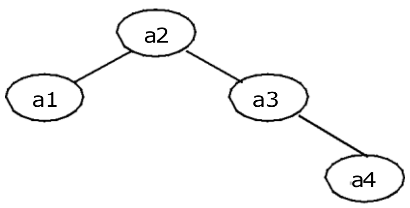
The left and right sub trees for T24 are T22 and T34 respectively.

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} -- 1

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

Fi (y) = max {fi-1 (y), fi-1 (y - wi) + pi} – 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, $0 \leq y \leq m$. 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 $0 < y \leq m$. 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), $y \geq yk$; and fi (y) = fi (yj), $yj \leq y \leq yj+1$. So, we need to compute only fi (yj), $1 \leq j \leq k$. We use the ordered set S^i = {(f (yj), yj) | $1 \leq j \leq k$} 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) | (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 $Pj \leq Pk$ 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 $\leq$ i $\leq$ 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 $1 - (1 - r_i)^{mi}$.

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 ~
$1 < i < n$

Subject to ~ $Ci\ mi < C$
$1 \leq i < n$

mi $\geq$ 1 and interger, 1 $\leq$ i $\leq$ n

Assume each Ci > 0, each mi must be in the range 1 $\leq$ mi $\leq$ ui, where

ui ~ ~ ~C +Ci n ~ C ~ J ~ / C~ U
ILk

The upper bound ui follows from the observation that mj $\geq$ 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 $\prod_{1 \leq j \leq i} q\$ m_J$

Subject to the constrains:

$\sum_{1 \leq i \leq i} C_J\, m_J \sim x$ and 1 $\leq$ mj $\leq$ uJ, 1 $\leq$ j $\leq$ 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, $0 \leq x \leq C$ 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 *0* i (mi) =1 – $(1- ri)^{mi}$

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

$$u_i = \sim \lfloor C + C_i - \sum_{1}^{n} C_j \Big/ C_i \rfloor$$

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 useS -* 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

$$S1 = \{S^1_1, S^1_2\}$$

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

$$S2 = \{S^2_1, S^2_2, S^2_3\}$$

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

$$S3 = \{S^3_1, S^3_2, S^3_3\}$$

Now find S^1_1 (x), x

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

$\emptyset 1(1) = 1 - (1 - r_1)^{m1} = 1 - (1 - 0.9)^1 = 0.9$

$\emptyset 1(2) = 1 - (1 - 0.9)^2 = 0.99$

$S^1_1 = \{f1(x), x\} = \{0.9, 30\}$

$S^1_2 = \{0.99, 30 + 30\} = \{0.99, 60\}$

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

Next find $S^2_1 = \{f2(x), x\}$

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

~2 ~1~ ~ 1 ~~1 ~ rI ~ = 1 – (1 – 0.8) = 1 – 0.2 = 0.8
mi 1

~2~ ~ 1 ~~1 ~ 0.8~ 2 = 0.96

$\emptyset_2(3) = 1 - (1 - 0.8)^3 = 0.992$

= {(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 ~ 3 S^2_1 ~2~ ~ 1 ~ ~1 0.5~ 2 = 0.75

Ø ~ 3 S^2_2 ~3~ ~ 1 ~ ~1 0.5~ 3 = 0.875

S^3_1

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^3_2 ={(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 ≤ x ≤ C; 1 ~ $mn < un$

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

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

$u_i = \sim \left(C + C_i - \sim C_J \right) / C_i \sim$

Using the above equation compute u1, u2 and u3.

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

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

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

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

= max {3(1) f2(105 - 20), <u>63(2) f2(105 - 20x2),</u> ~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.

f2 (85) = max {2 (m2). f1 (85 -15m2)} $1 ! m2 ! u2$

= 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)}

$= \max \{\sim1(1) \times 1, {}_{t1(2)} \times 1\} = \max \{0.9, 0.99\} = 0.99$

$f1\ (55) = \max \{t1(m1).\ f0(55 - 30m1)\}$
$1 ! m1 ! u1$

$= \max \{\sim1(1)\ f0(50 - 30), t1(2)\ f0(50 - 30x2)\}$

$= \max \{\sim1(1) \times 1, {}_{t1(2)} \times -oo\} = \max \{0.9, -oo\} = 0.9$

$f1\ (40) = \max \{\sim1(m1).\ f0\ (40 - 30m1)\}$
$1 ! m1 ! u1$

$= \max \{\sim1(1)\ f0(40 - 30), t1(2)\ f0(40 - 30x2)\}$

$= \max \{\sim1(1) \times 1, {}_{t1(2)} \times -oo\} = \max\{0.9, -oo\} = 0.9$

$f2\ (65) = \max \{2(m2).\ f1(65 -15m2)\}$
$1 ! m2 ! u2$

$= \max \{2(1)\ f1(65 - 15), \underline{62(2)\ f1(65 - 15x2),} \sim2(3)\ f1(65 - 15x3)\} = \max \{0.8\ f1(50),$ $0.96\ f1(35), 0.992\ f1(20)\}$

$= \max \{0.8 \times 0.9, 0.96 \times 0.9, -oo\} = 0.864$

$f1\ (50) = \max \{\sim1(m1).\ f0(50 - 30m1)\}$
$1 ! m1 ! u1$

$= \max \{\sim1(1)\ f0(50 - 30), t1(2)\ f0(50 - 30x2)\}$

$= \max \{\sim1(1) \times 1, {}_{t1(2)} \times -oo\} = \max\{0.9, -oo\} = 0.9$ $f1\ (35) = \max$

$\sim1(m1).\ f0(35 - 30m1)\}$

$1 ! m1 ! u1$

$= \max \underline{\{\sim1(1).f0(35-30),} \sim1(2).f0(35-30x2)\}$

$= \max \{\sim1(1) \times 1, {}_{t1(2)} \times -oo\} = \max\{0.9, -oo\} = 0.9$

$f1\ (20) = \max \{\sim1(m1).\ f0(20 - 30m1)\}$
$1 ! m1 ! u1$

$= \max \{\sim1(1)\ f0(20 - 30), t1(2)\ f0(20 - 30x2)\}$

$= \max \{\sim1(1) \times -, \sim1(2) \times -oo\} = \max\{-oo, -oo\} = -oo$

$f2\ (45) = \max \{2(m2).\ f1(45 -15m2)\}$
$1 ! m2 ! u2$

$= \max \{2(1)\ f1(45 - 15), \sim2(2)\ f1(45 - 15x2), \sim2(3)\ f1(45 - 15x3)\} = \max \{0.8\ f1(30),$

$0.96\ f1(15), 0.992\ f1(0)\}$

$= \max \{0.8 \times 0.9, 0.96 \times -, 0.99 \times -oo\} = 0.72$

f1 (30) = max {~1(m1). f0(30 - 30m1)} 1<m1~u1

= 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.