UNIT IV:

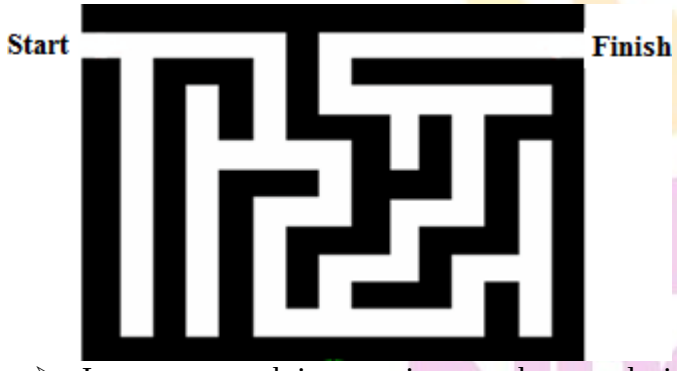
Backtracking: General method, applications-n-queen problem, sum of subsets problem, graph coloring, Hamiltonian cycles.
Branch and Bound: General method, applications - Travelling sales person problem,0/1 knapsack problem- LC Branch and Bound solution, FIFO Branch and Bound solution.

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:

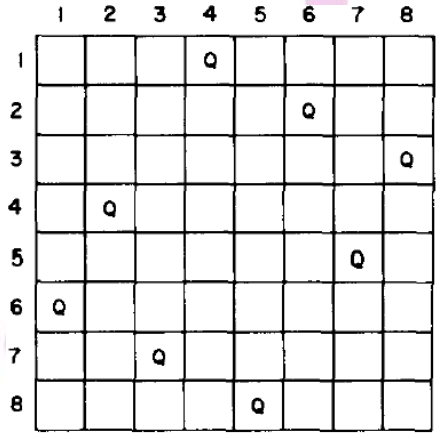
- Explicit constraints.
- Implicit constraints.

Explicit constraints: Explicit constraints are rules that restrict each x_i to take on values only from a given set.
Example: $x_i \geq 0$ 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 X_i must relate to each other.
Applications of Backtracking:

- N Queens Problem
- Sum of subsets problem
- Graph coloring
- Hamiltonian cycles.

N-Queens Problem:

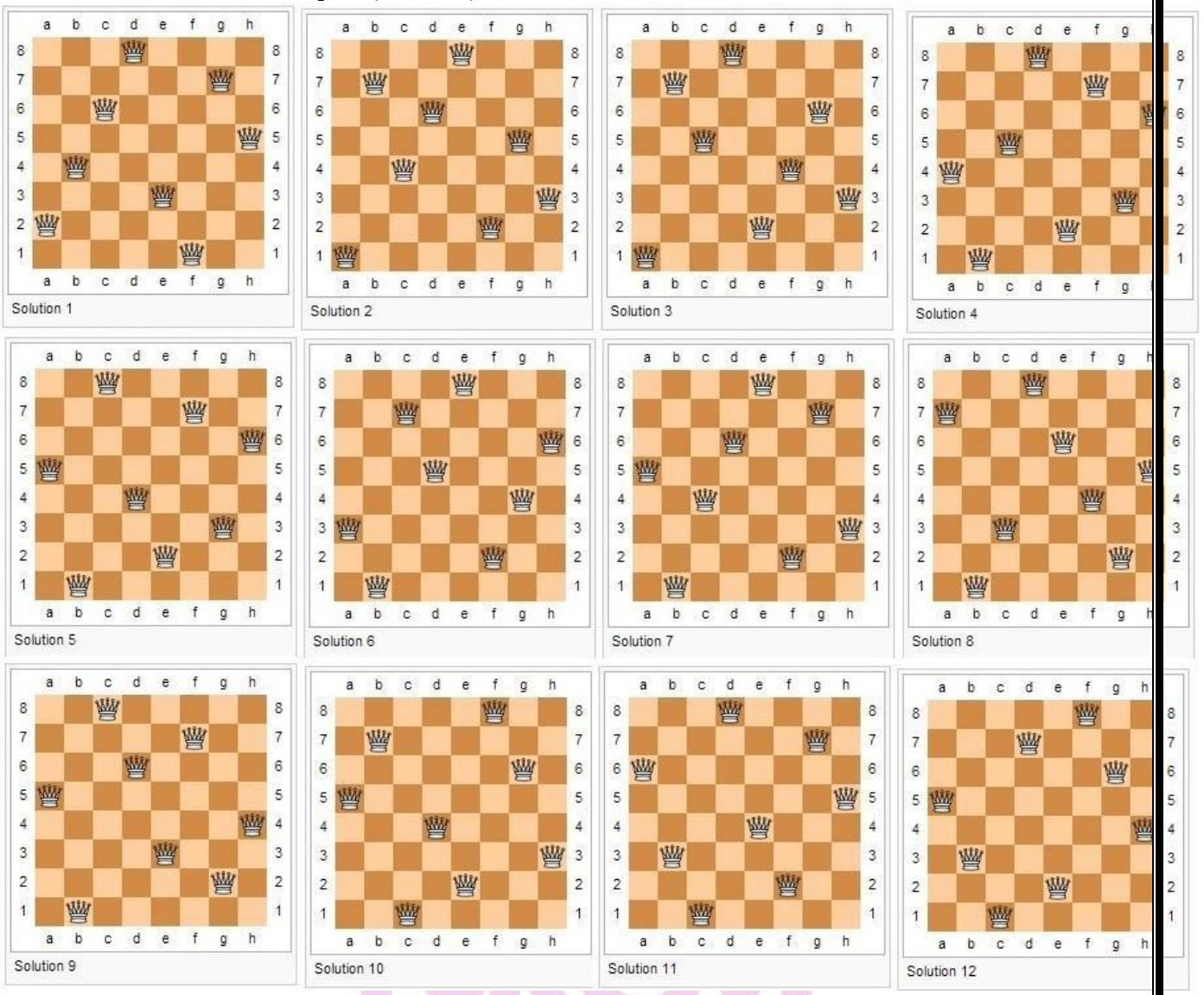
It is a classic combinatorial problem. The eight queen's puzzle is the problem of placing eight queens puzzle is the problem of placing eight queens on an 8×8 chessboard so that no two queens attack each other. That is so that no two of them are on the same row, column, or diagonal.
The 8-queens puzzle 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)

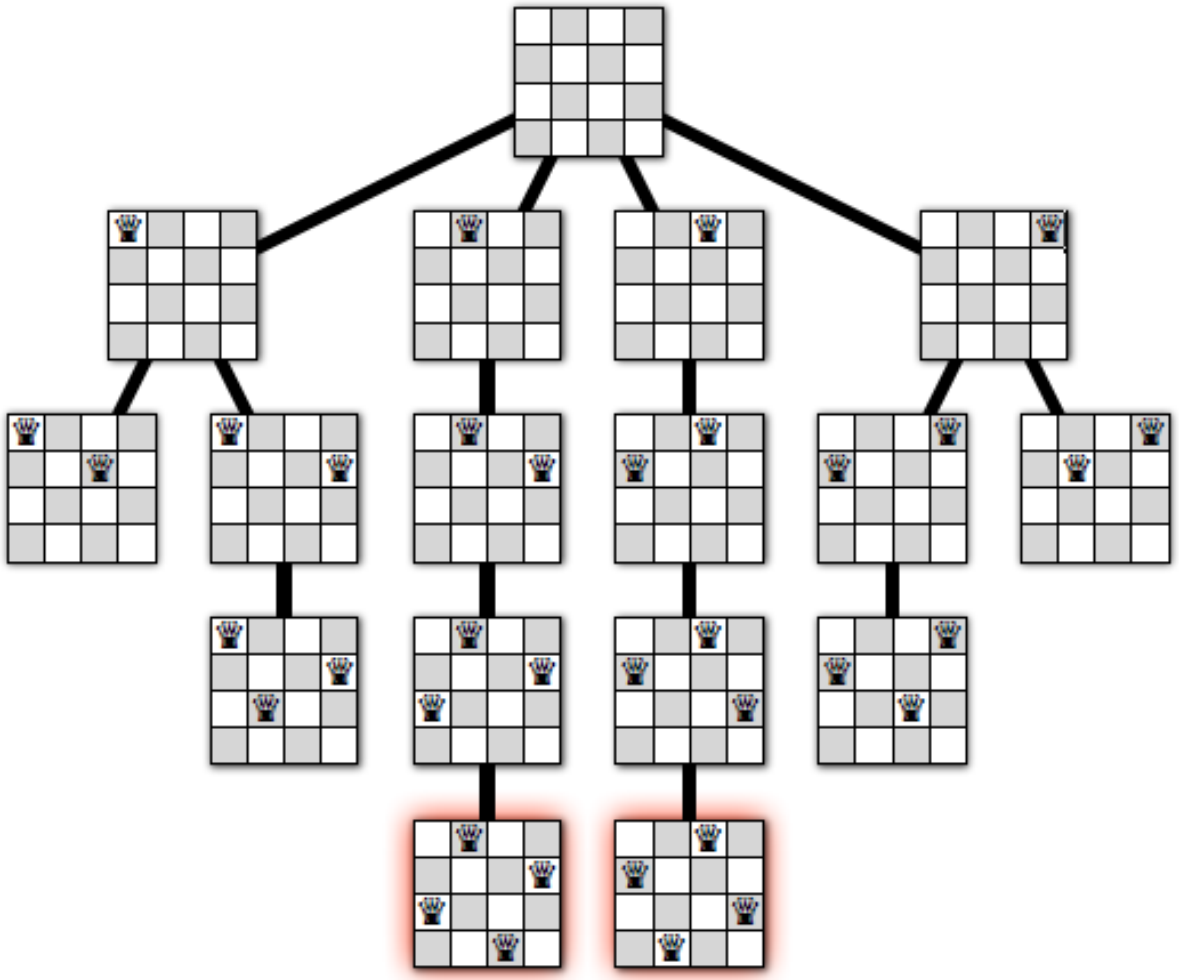
In the same way for n-queens are to be placed on an n×n chessboard, the solution space consists of all n! Permutations of n-tuples (1,2, ---n).



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 $1 \le i \le n$, & 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) $1 \le k \le n$, different solutions may have different sized tuples.

- Explicit constraints requires $x_i \in$ {j / j is an integer $1 \le j \le n$ }
- 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 $x_i < x_{i+1}$ $1 \le i \le k$

W_i → 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.

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

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.

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

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

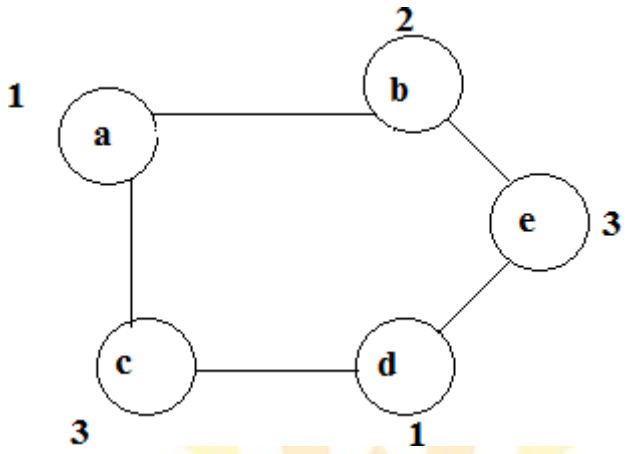
Recursive backtracking algorithm for sum of subsets problem

```
Algorithm SumOfSub(s, k, r)
{
//s = Σ_{j=1}^{k-1} W(j)X(j) and r = Σ_{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.

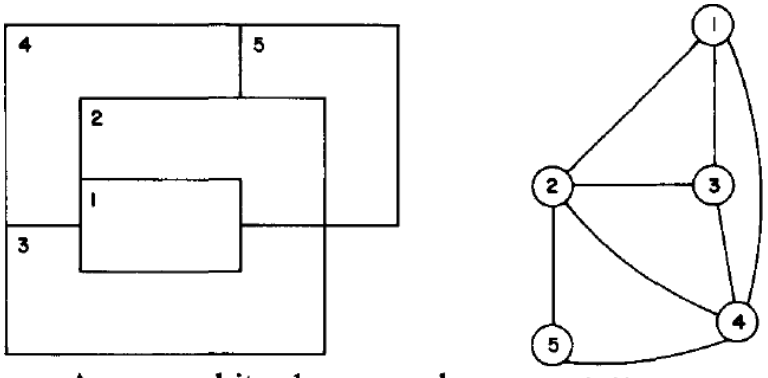
Example



- 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**

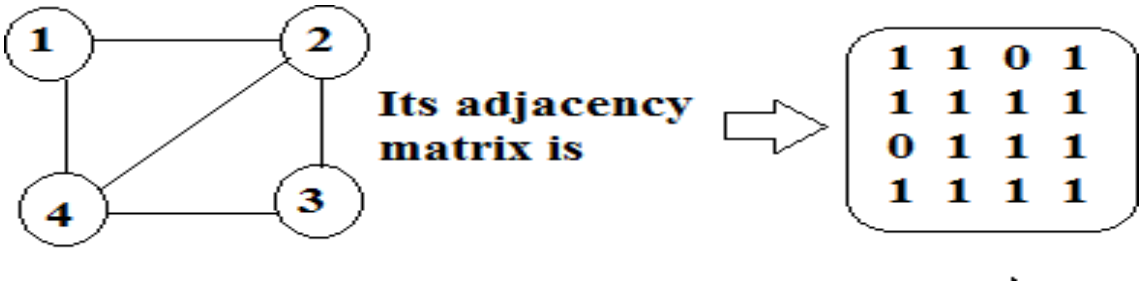
The above map requires 4 colors.

- 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.

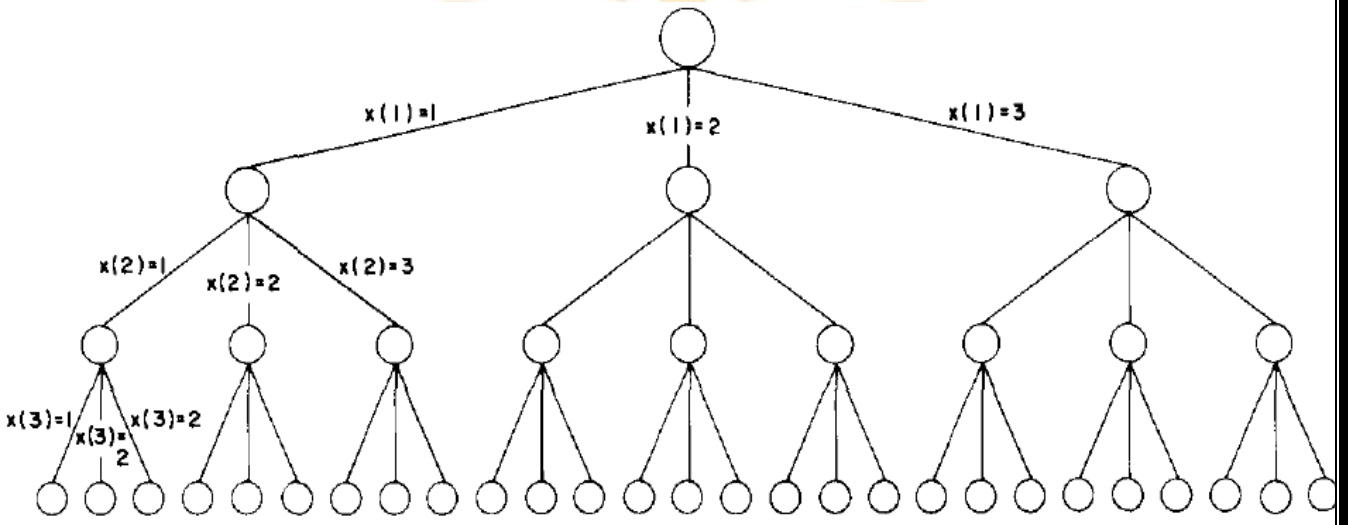
Suppose we represent a graph by its adjacency matrix G[1:n, 1:n]

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



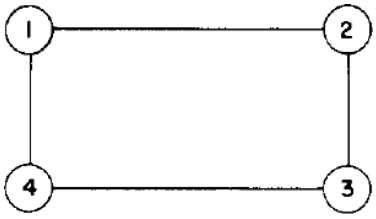
State space tree for MCOLORING when $n = 3$ and $m = 3$

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

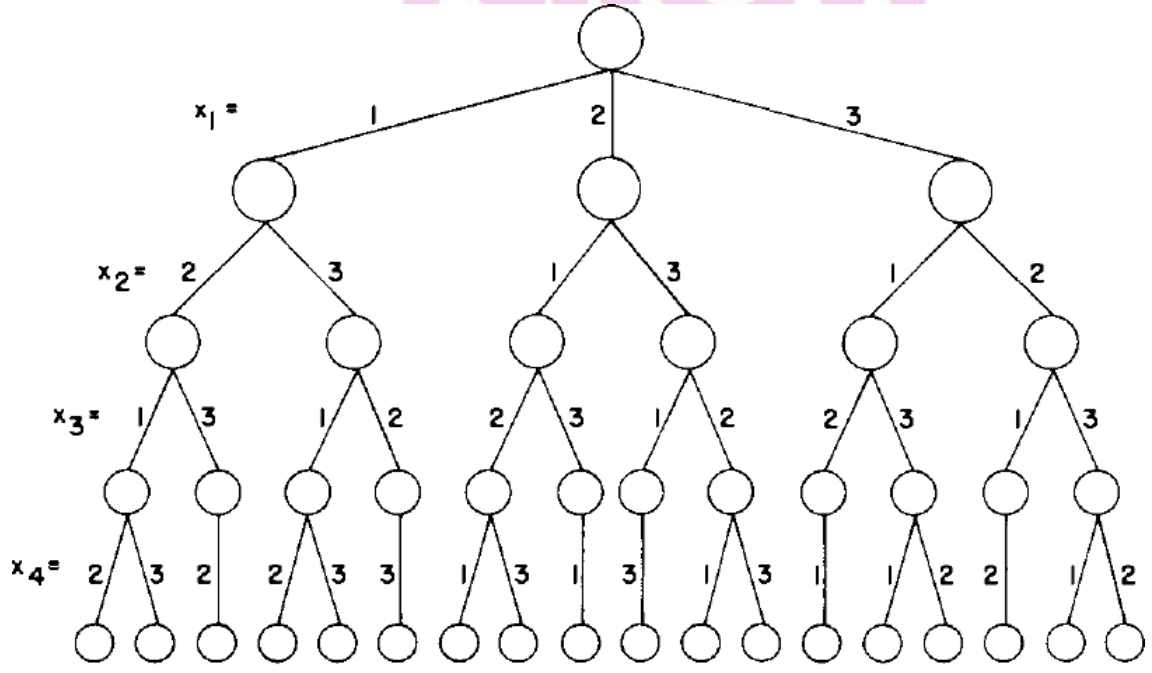
Algorithm mColoring(k){
// g(1:n, 1:n)→ boolean adjacency matrix.
// k→index (node) of the next vertex to color.
repeat{
nextvalue(k); // assign to x[k] a legal color.
if(x[k]=0) then return; // no new color possible
if(k=n) then write(x[1: n];
else mcoloring(k+1);
}
until(false)
}
	Algorithm NextValue(k){
	//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

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

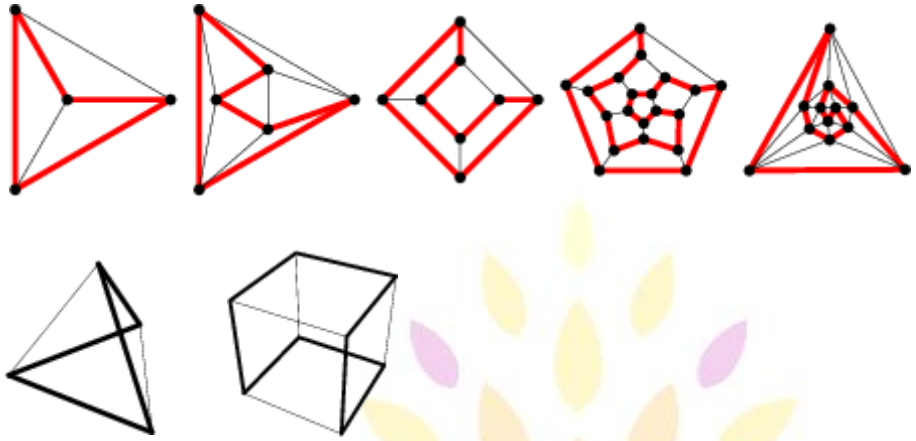


A 4 node graph and all possible 3 colorings

NRCM
your roots to success.

Hamiltonian Cycles:

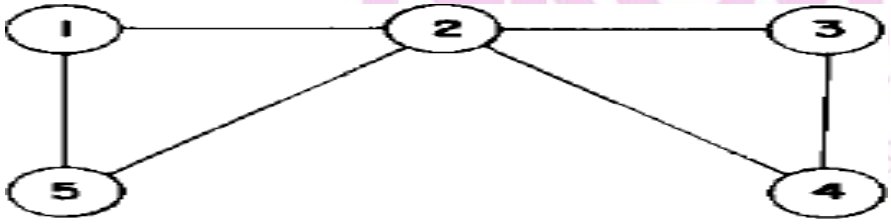
- **Def:** Let G=(V, E) be a connected graph with n vertices. A Hamiltonian cycle is a round trip path along n-edges of G that visits every vertex once & returns to its starting position.
- It is also called the Hamiltonian circuit.
- Hamiltonian circuit is a graph cycle (i.e., closed loop) through a 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 $v_1, v_2, \text{---} v_{n+1}$, then the edges (v_i, v_{i+1}) are in E, $1 \leq i \leq n$.

→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 1st 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**

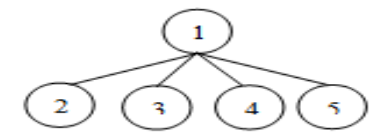
Algorithm NextValue(k)
{
// x[1: k-1]→ is path of k-1 distinct vertices.
// if x[k]=0, then no vertex has yet been assigned to x[k]
Repeat{
X[k]=(x[k]+1) mod (n+1); //Next vertex
If(x[k]=0) then return;
If(G[x[k-1], x[k]]≠0) then
{
For j:=1 to k-1 do if(x[j]=x[k]) then break;
//Check for distinctness
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

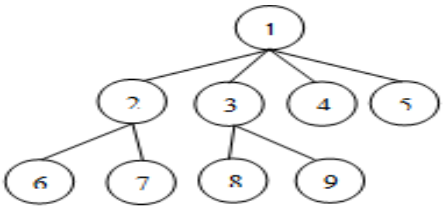
Branch & Bound (B & B) is general algorithm (or Systematic method) for finding optimal solution of various optimization problems, especially in discrete and combinatorial optimization.

- 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.

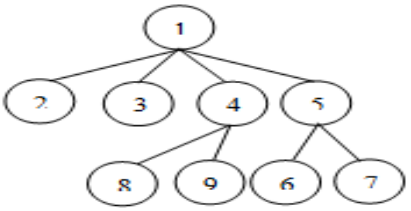
✓ **Dead node→** is a generated node that is not to be expanded or explored any further. All children of a dead node have already been expanded.



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

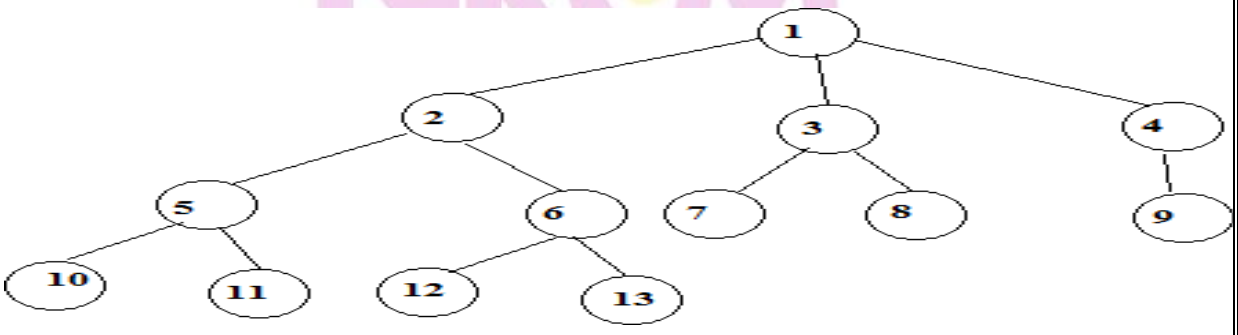
1) FIFO (First In First Out) search
2) LIFO (Last In First Out) search
3) LC (Least Count) search

FIFO B&B:

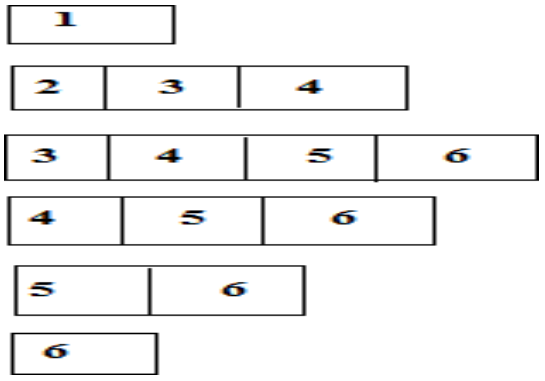
FIFO Branch & Bound is a BFS.
In this, children of E-Node (or Live nodes) are inserted in a queue.
Implementation of list of live nodes as a queue

✓ 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'



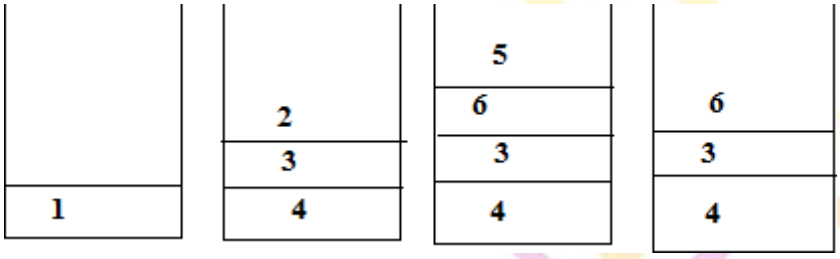
LIFO B&B:

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:

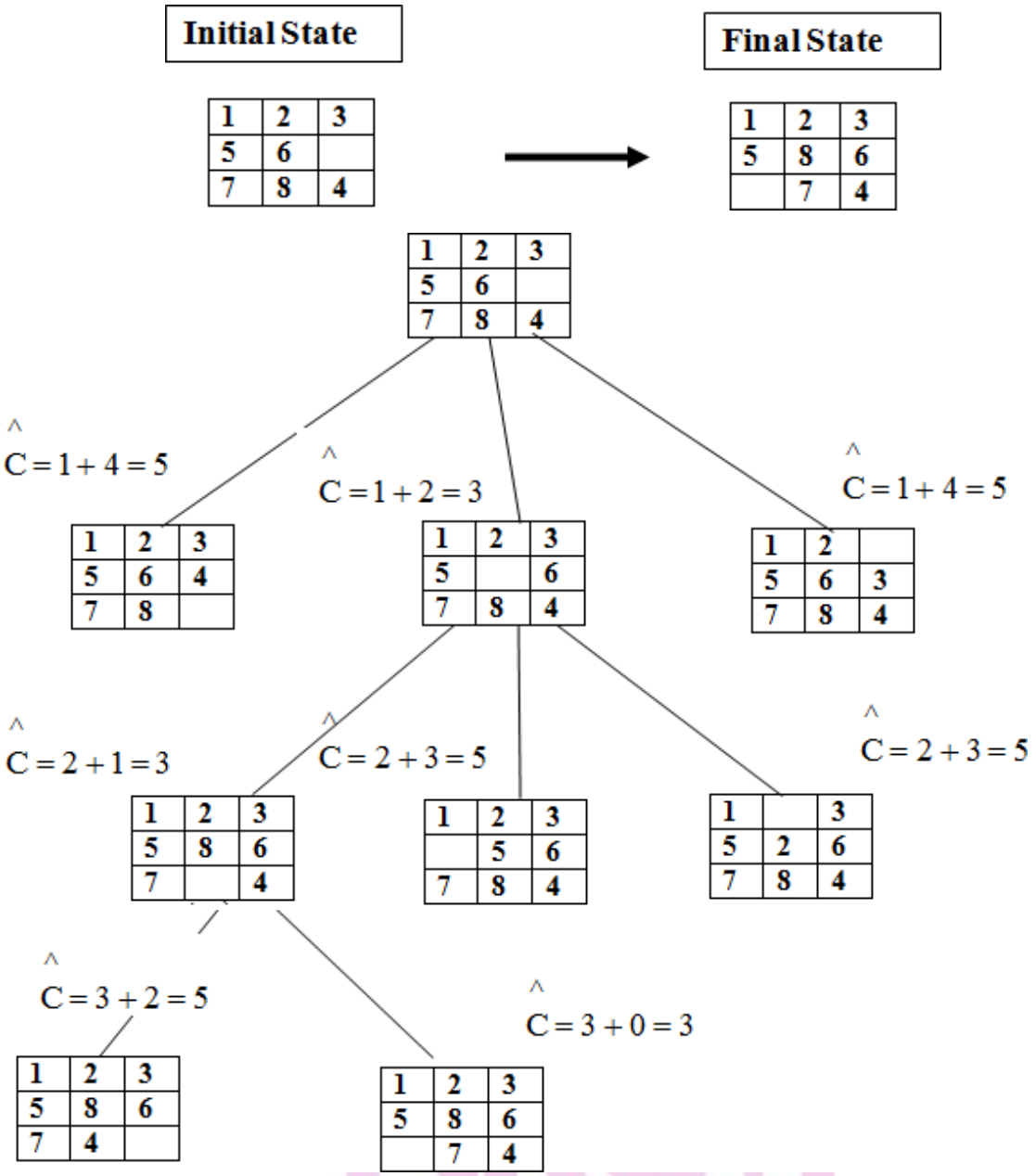
8-puzzle

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} → cost of edge <i, j>

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

$|V|=n$ → total number of vertices.

Assume that every tour starts & ends at vertex 1.

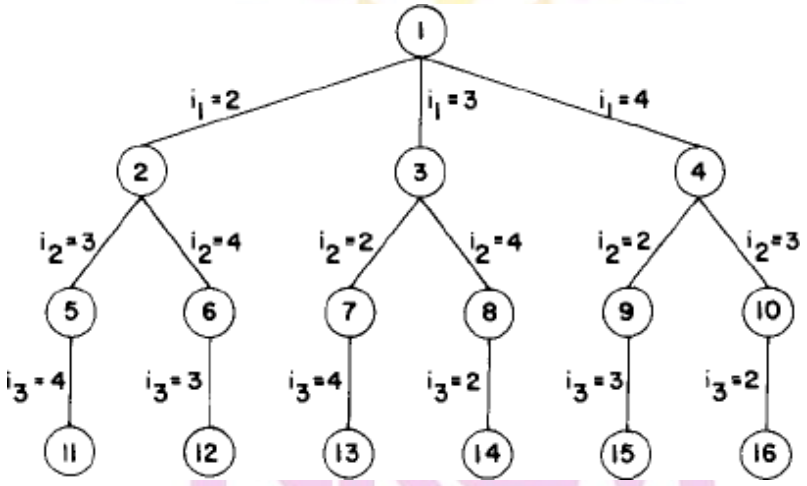
Solution Space S= {1, Π , 1 / Π is a permutation of (2, 3. 4 -----n) } then $|S|=(n-1)!$

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.

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

TSP is solved by using LC Branch & Bound:

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 $\hat{C}(r) \le C(r) \le u(r)$ → for all nodes r.

Cost C(.)→ is the solution node1 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 $\hat{C}(r) \rightarrow$ 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: **Node 1**
- Reduced Matrix: To get the lower bound of the path starting at node 1

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

$\begin{bmatrix} inf & 10 & 20 & \\ 13 & inf & 14 & \\ 1 & 3 & inf & \\ 16 & 3 & 15 & in \\ 16 & 4 & 7 & 16 \end{bmatrix}$	$\begin{bmatrix} inf & 10 & 20 & \\ 13 & inf & 14 & \\ 1 & 3 & inf & \\ 16 & 3 & 15 & in \\ 12 & 0 & 3 & 12 \end{bmatrix}$	$\begin{bmatrix} inf & 10 & 20 & \\ 12 & inf & 14 & \\ 0 & 3 & inf & \\ 15 & 3 & 15 & in \\ 11 & 0 & 3 & 12 \end{bmatrix}$
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

- **Choose to go to vertex 3: Node 3**

  - 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:

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

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

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

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

- Choose to go to vertex 4: **Node 4**

  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:

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

  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

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

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

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

- **Choose to go to vertex 3: Node 7 (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

- Choose to go to vertex 5: **Node 8** (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

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

- **Choose to go to vertex 3: Node 9 (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

- **Choose to go to vertex 5: Node 10 (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

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

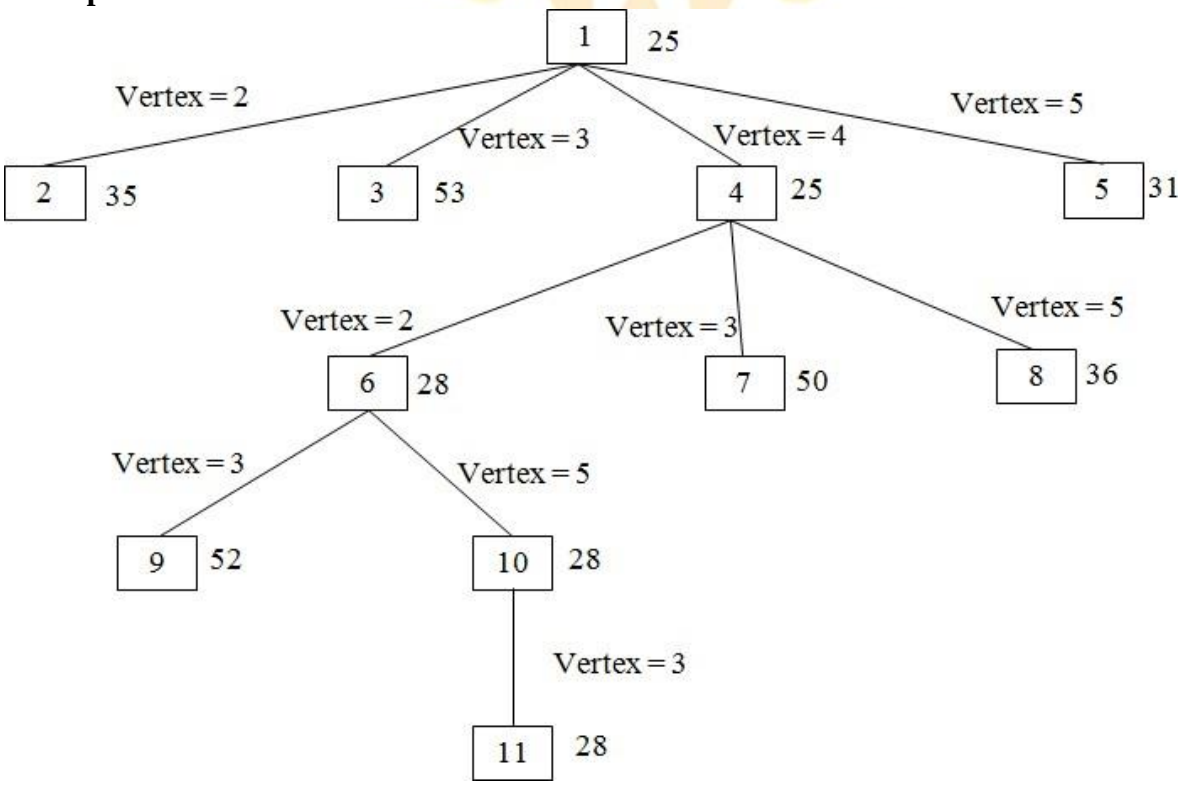
- **Choose to go to vertex 3: Node 11 (path is 1->4->2->5->3)**

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, Z_1 **to** Z_n where Z_i has value P_i & weight 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

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

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

X_i**=0 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 cˆ(x) and u(x) such that for every node x,

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

- Computing c^(·) and u(·)

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;
}
```