2/3/19

UNIT-IV
BACKTRACKING

General method -

(It is a backtracking method that the desi)x

In the backtracking method:

① the desired solution is expressible as an 'n' tuple $(x_1, x_2 \cdots x_n)$ where x_i is choosen from some finite set S_i.

② The solution maximizes or minimizes or satisfies a criterian function $c(x_1, x_2 \cdots x_n)$.

③ The problem can be categorized into 3 categories.

For example - A problem P, c be the set of constraints for the problem P, let D be the set containing all solutions satisfying c, then

① finding whether there is any feasible soln is called decision problem.

② what is the best soln is: the optimization problem.

③ Listing of all the feasible soln is: enumeration problem.

④ The Basic idea of backtracking is to Build-up a vector, one component at a time and to test whether vector being formed has any chance of success.

⑤ The major advantage of this algorithm is that can realise the fact that the partial vector generated will not lead to an optimal solution. In such situation the vector can be ignored.

⑥ Backtracking algorithm determines the solution by systematically searching the solution space. (feasible soln) for the given problem.

⑦ Backtracking is a DFS with some Bounding function. All solutions using Backtracking are required to satisfy a complex set of constraints. The constraints may be explicit or Implicit.

⑧ Explicit constraints are rules which restricts each vector element to be choosen from the given set.

Implicit constraints are rules which determine which of the tuples in the solution space actually satisfy the criterian function.

5/3/19 Terminology used in Backtracking -

→ Back tracking Algorithms determine problem solutions by Systematically searching for the solutions using Tree structure.

→ Each node in the tree is called a Problem state.

→ All paths from root node to other nodes define the statespace of the problem.

→ The solution states are those problem states 's' for which the path from root to s defines the tuple in the solution state.

→ Answerstate -

These are the leaf nodes which correspond to an element in the set of solutions, these are the states which satisfy implicit constraints.

→ A node which is been generated and whose children have not yet been generated is called 'live node'.

→ The live node whose children are currently expanded is called E-node.

→ A dead node is a generated node which is not to be expanded further or all of children have been generated.

→ There are two methods of generating state search tree.

(1) Backtracking - In this method as soon as the new child 'c' of the current E-node N- is generated This child will be new E-node.

The N-will becomeeee the E-node again when the subtree 'c' has been fully explored.

(2) Branch and Bound - E-node will remain as E-node until it is dead.

→ Both Backtracking and Branch & Bound methods use bounding functions. The Bounding functions are used to kill live nodes without generating all their childrens. The care has to be taken while killing live nodes so that atleast one answer node or all answer nodes are obtained.

Applications of Backtracking-

1. 8 Queen's problem (n-Queen's problem)
2. sum of subset problem
3. Graph colouring problem
4. finding Hamiltonian cycle
5. knapsack Problem.

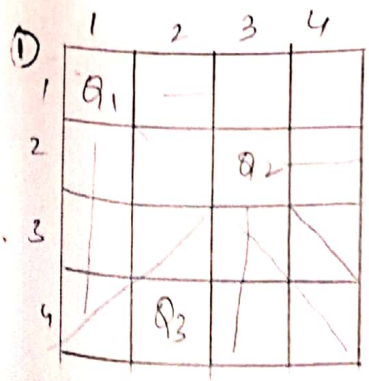
7/3/19

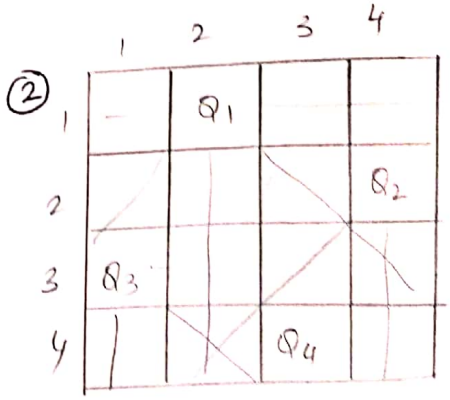
① n-Queen's problem -

The n-Queen's problem can be stated as nxn chessboard on which n Queen's has to be placed so that no two queen's attack each other by being in the same row or in the same column or in the diagonal.

4-QUEEN's PROBLEM-

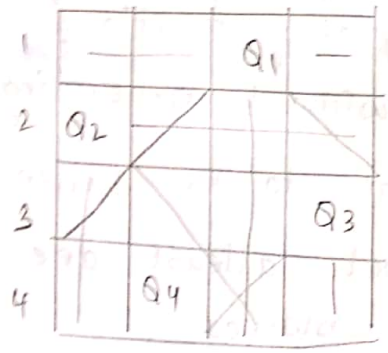
→ 4-Queen's problem can be solved as the following.

① 

② 

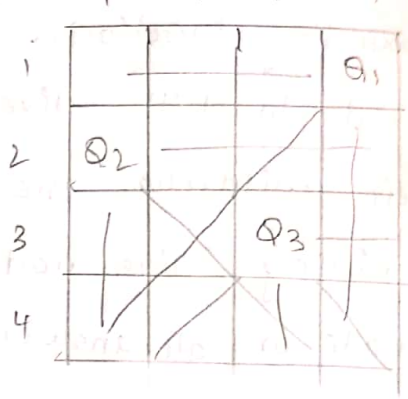
(✓)

③



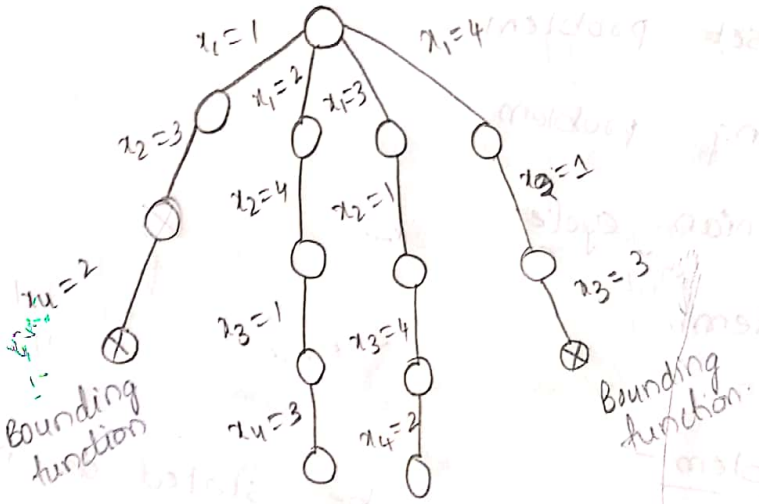
(✓)

④

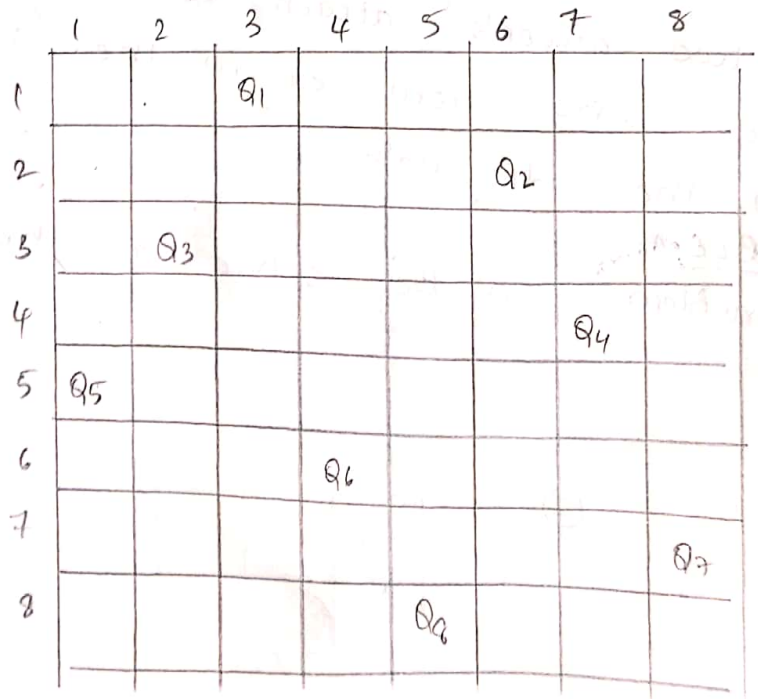


(X)

State Space tree -



8-QUEEN'S PROBLEM -



State Space tree -

[10 total problems]

Algorithm -

```
Algorithm Queen(n)
// Problem description: This algorithm is for implementing
// n-queen's problem.
// Input: total number of queen's n.
      for column ← 1 to n do
      {
if (place (row, column)) then  //checking queens on same
                                 diagonals or not.
      {
 board[row] = column
 if (row = n) then             //dead end
  print_board (n)              //printing board configuration
   else
   Queen (row+1, n)
     }
     }
Algorithm place (row, column)
// problem description: This algorithm is for placing the queen
//at appropriate place.
//Input: row and column of the chessboard.
//output: returns 0 for the conflicting row and
//column position and 1 for no conflict.

for. i ← 1 to row-1 do
 {
  if (board[i] = column) then
  return 0
  else if (abs (board[i] - column) = abs (i - row)) then
   return 0
    }
    return 1.
```

(2) Sum of subset problem -

Problem statement - Let $S = \{s_1, s_2, \ldots s_n\}$ be a set of "n" positive Integers, then find a subset whose sum is equal to given positive integer "d".

→ It is always convenient to sort the set's elements in ascending order, ie, $s_1 \leq s_2 \leq \ldots \leq s_n$.

Step1 -

→ let 's' be a set of elements and 'd' is the ~~expected~~ expected sum. Then start with an empty set.

Step-2 -

Add to the subset the next element from the list.

Step-3 -

If the subset is having sum 'd' then stop with that subset as solution.

step-4 -

If the subset is not feasible or if reached the end of the set then backtrack through the subset until to find the most suitable soln.

Step-5 -

If the subset is feasible then repeat step-2.

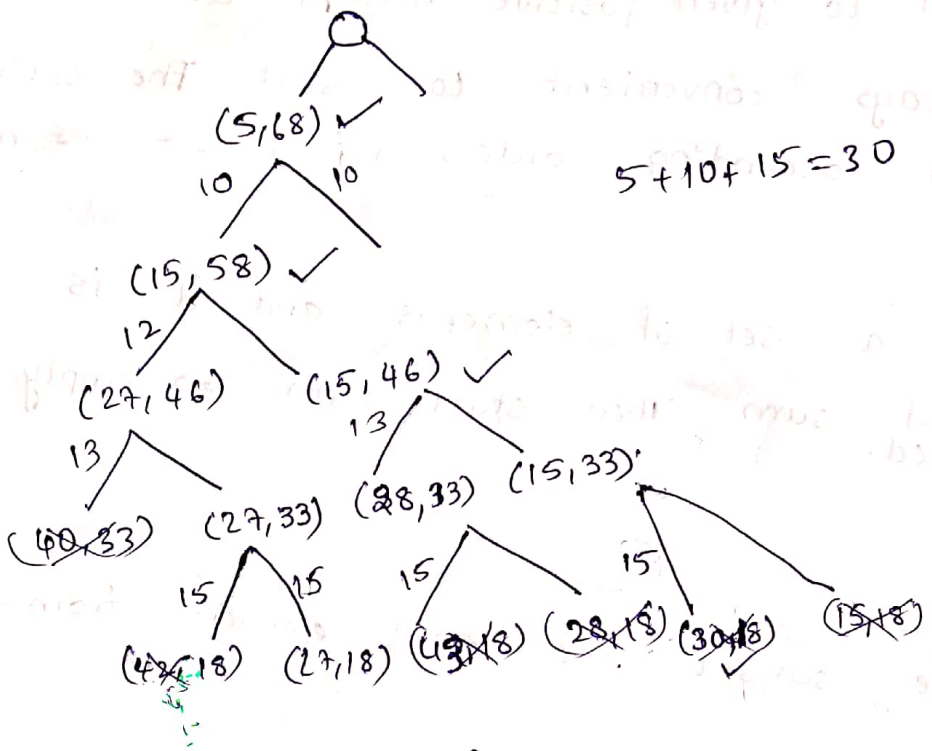
Step-6 -

If all the elements are visited without finding a suitable subset and if no backtracking is possible then stop without soln.

State Space tree-

$S = \{5, 10, 12, 13, 15, 18\}$ $d = 30$

Sum = 73.



$5 + 10 + 15 = 30$

→ $S = \{5, 10, 12, 13, 15, 18\}$ $d = 30$

Initially subset = {}	sum = 0	
5	5	add next element
5, 10	15, 15 < 30	add next element
5, 10, 12	27, 27 < 30	add next element
5, 10, 12, 13	40, 40 > 30	Backtrack, 13
5, 10, 12, 15	42, 42 > 30	Backtrack, 15
5, 10, 12, 18	45, 45 > 30	Backtrack, 18
5, 10, 13 *	28	add next element
5, 10, 13, 15	43, 43 > 30	Backtrack
5, 10, 13, 18	46, 46 > 30	Backtrack
5, 10, 15	30	Solution obtained sum = 30 = d.

Algorithm –

```
Algorithm  Sum_subset (Sum, index, Remaining_Sum)
a [index] ← 1
if (sum+w [index] = d) then
write (a[1 .... index])
else if (sum + w [index] + w [index+1] > d) then
sum_subset ((sum + w [index]), (index + 1), Remaining_sum-
                                              w[index]));
if (sum + Remaining - sum - w[index] ≥ d) AND
      (sum + w[index+1] ≤ d) then
{
a [index] ← 0
sum_subset (sum, (index+1), (Remaining_Sum - w[index]));
}
```

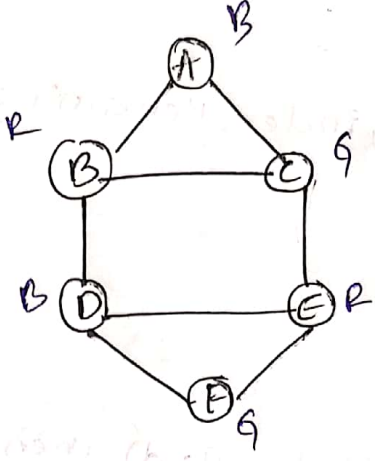
11/3/19

Graph colouring –

Graph colouring is a problem of colouring each vertex in graph in such a way that no two adjacent vertices have same colour and 'm' colours are used. This problem is also called m-colouring problem.

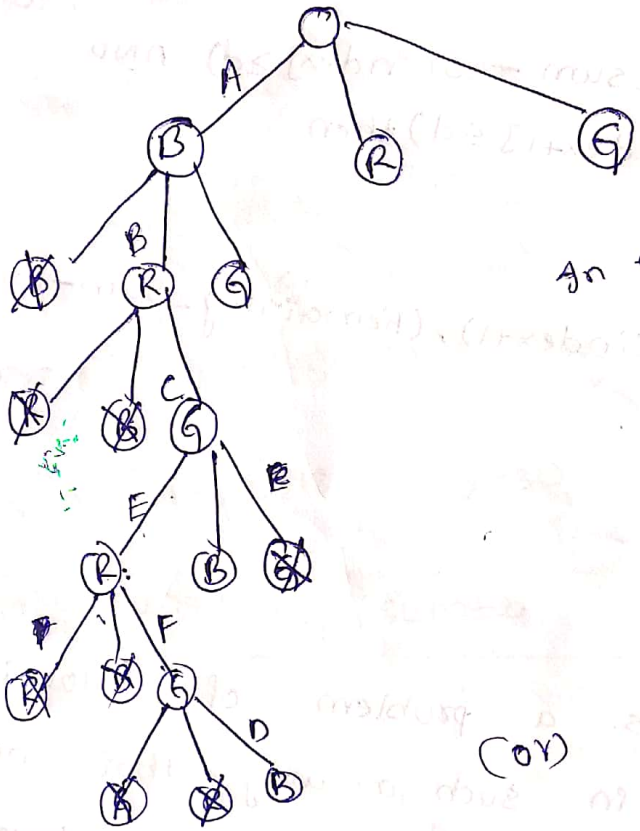
→ If the degree of given graph is 'd' then can colour it with 'd+1' colours. The least number of colours needed to colour the graph is called its chromatic number.

Ex-



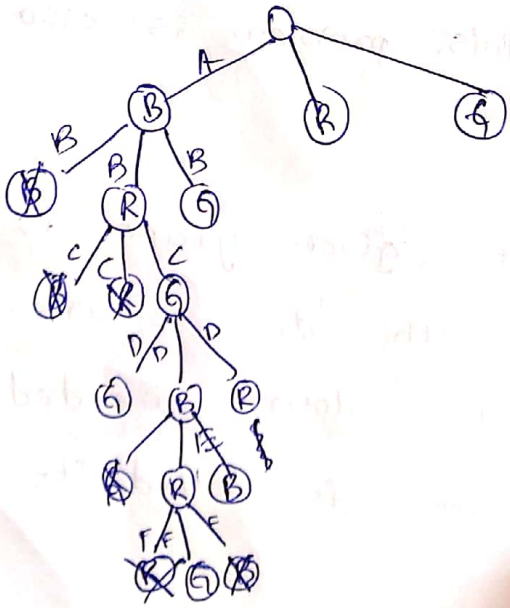
colours - B, R, G

State Space tree -



An exam total

(OR)



~~Step 1~~ - step

A graph G consisting of vertices from A to F there are 3 colours Red, Green, Blue have to be used to colour the graph.

→ Step-1 - Initially the node 'A' has been chosed to fill with the colour Blue. The node 'B' and 'C' are adjusted to 'A'. So, these two are not coloured with Blue. So, node 'B' is coloured with Red and node 'C' is adjacent to A and B it is coloured with Green.

Step-2 - Repeat the same for node D,E,F and those vertices has to be coloured in such a way that the adjacent vertices should not have same colour.

12/3/19

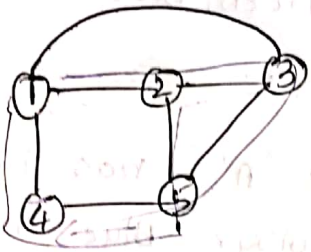
Hamiltonian cycle -

→ In an undirected connected graph if there is a cycle exists visiting all the nodes exactly once i.e, if node 'x' is starting vertex and the node 'y' is ending vertex, there exists a path from x to y visiting the nodes only once and reach the 'y' from 'x'.

→ If there is a articulation point in the graph then hamiltonian cycle is not possible.

→ If there exists a pendent vertexes in a graph then hamiltonian cycle is not possible.

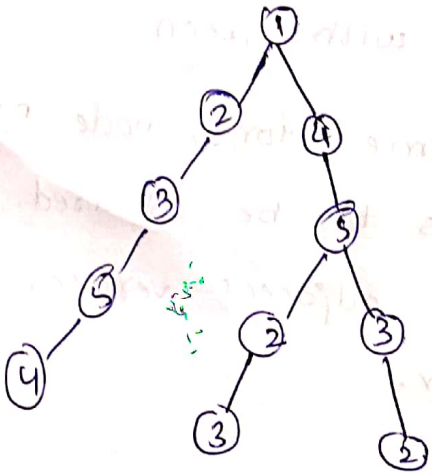
Ex -



	1	2	3	4	5
1	0	1	1	1	0
2	1	0	1	0	1
3	1	1	0	0	1
4	1	0	0	0	1
5	0	1	1	1	0

State Space tree -

1	2	3	5	4	1

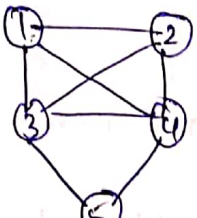


1-2-3-5-4

1-4-5-2-3

1-4-5-3-2

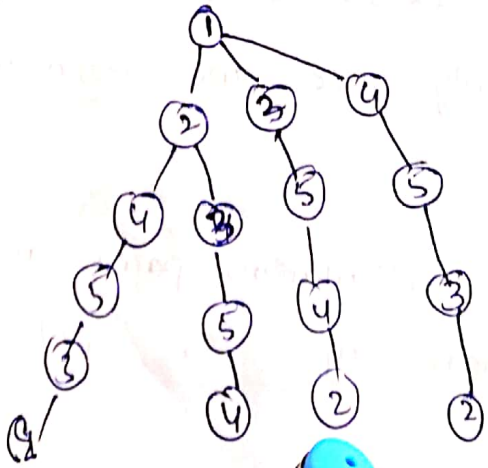
(2)



1	2	4	5	3	1

	1	2	3	4	5
1	0	1	1	1	0
2	1	0	1	1	0
3	1	1	0	1	1
4	1	1	1	0	1
5	0	0	1	1	0

State Space tree -



1-2-3-5-4-1

1-2-4-5-3-1

1-3-5-4-2-1

1-4-5-3-2-1

14/3/19

Branch and Bound -

Branch & Bound is a general algorithmic method for finding optimal solutions of various optimization problems.

→ Branch and Bounding method is a general optimization technique that applies where the greedy method and dynamic programming fails.

General method -

① In Branch and Bound method a state space tree is built and all children ~~of~~ (E-node) (~~are generated~~) of a live node are generated before any other node become a live node.

② In Branch and Bound technique a state space tree like BFS is called FIFO. Search. This is because the live node is FIFO.

③ The state space search like DFS or D-Search will be called LIFO search Because the live node is LIFO.

④ In this method a state space tree of possible solutions is generated. Then partitioning is called Branching, is done at each node of the tree. Then compute lower Bound

and Upper Bound at each node.

⑤ This computation leads to selection of answer nodes

⑥ Bounding Functions are used to avoid the generation of subtrees that do not contain answer node.

Algorithm →

```
Algorithm Branch_Bound()
{
// E- is a node pointer
  E← new (node)
 // start node
 // H- is heap for all the live nodes.
 while(true)
  {
 if (E- is a final leaf) then
  {
 write (path from E to the root);
 return ;
 }
 Expand(E);
 if (H is empty) then
 {
 write (there is no solution);
 return ;
 }
 E← delete_top (H);
 }.
 }
```

Algorithm Expand(E).

{

Generate all the children of E;

compute the approximate cost value of each child;

Insert each child into the heap H;

}

15/3/19

0/1 Knapsack problem -

→ To use the Branch and Bound technique to solve any problem it is first necessary to conceive of a state space tree for the problem. It can't be directly applied on Branch and Bound techniques to solve this problem as this Knapsack problem is maximization problem and the Branch and Bound techniques are minimization problems.

→ To overcome this difficulty by replacing the objective function $\sum p_i x_i$ ~~by~~ ~~to~~ with the function $-\sum p_i x_i$.

→ It is clear that $\sum p_i x_i$ is maximized iff $-\sum p_i x_i$ is minimized. This modification in the knapsack problem can be stated as

minimized $\left(-\sum_{i=1}^{n} p_i x_i\right)$ subjected to $\sum_{i=1}^{n} w_i x_i \le m$.

$x_i = 0$ or 1, $1 \le i \le n$.

→ This will be continued by assuming a fixed tuple size formulation for the solution space.

→ Every leaf node in the state space tree Representing an assignment for which $\sum_{1 \le i \le n} w_i x_i \le m$. is an answer node. All other leaf nodes are infeasible.

→ For a min. cost answer node to correspond to any optimal solution need to define

$c(x) = \sum_{1 \le i \le n} p_i x_i$ for every answer node 'x'.

The cost $c(x) = \infty$ for infeasible leaf nodes.

For non-leaf nodes c(x) is recursively defined to be min{c(lchild(x)), c(rchild(x))}
(lchild(x): left; rchild(x): right.)

→ Now we needed two functions $\hat{c}(x), u(x)$ such that $\hat{c}(x) \le c(x) \le u(x)$. for every node 'x'.
($\hat{c}(x)$: lower; $u(x)$: upper)

Algorithm –

```
Algorithm  UBound (cp, cw, k, m)
{
  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;
}

16/3/19

LC Branch and Bound (LCBB) Solution for 0/1 -

$n=4$
$m=15$

$w = \{2, 4, 6, 9\}$

$p = \{10, 10, 12, 18\}$

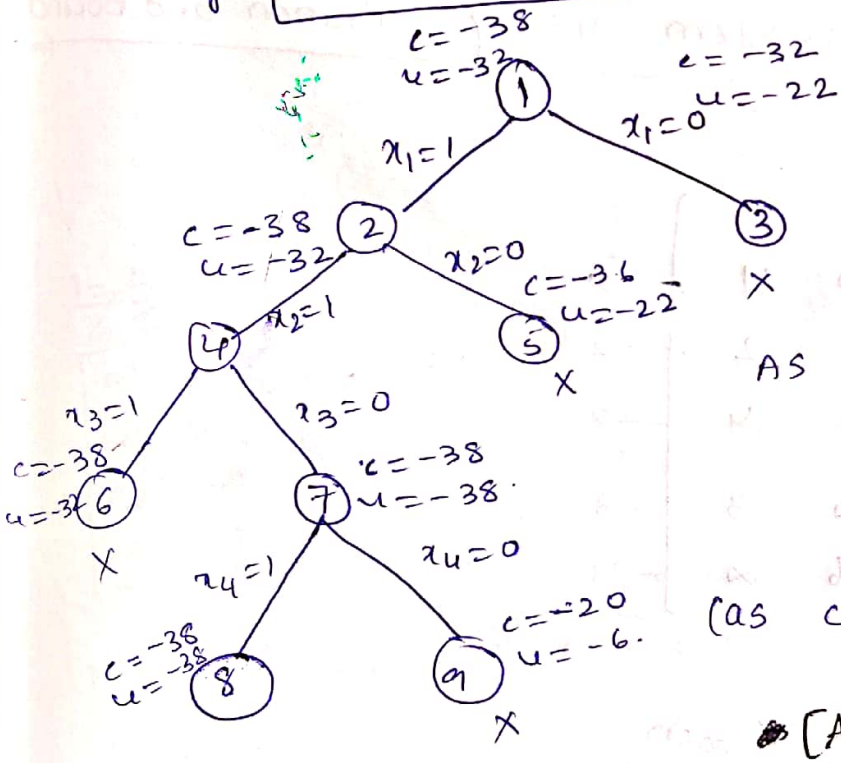
p	10	10	12	18
w	2	4	6	9

$u = -\sum p_i x_i$ (No fractions)

$c = -\sum p_i x_i$ (Fractions allowed)

Initially $u = -38$

10	10	12	18
2	4	6	9



AS ($u=-38 < u=-22, u=-22$)

(as $c=-20 > c=-38$)

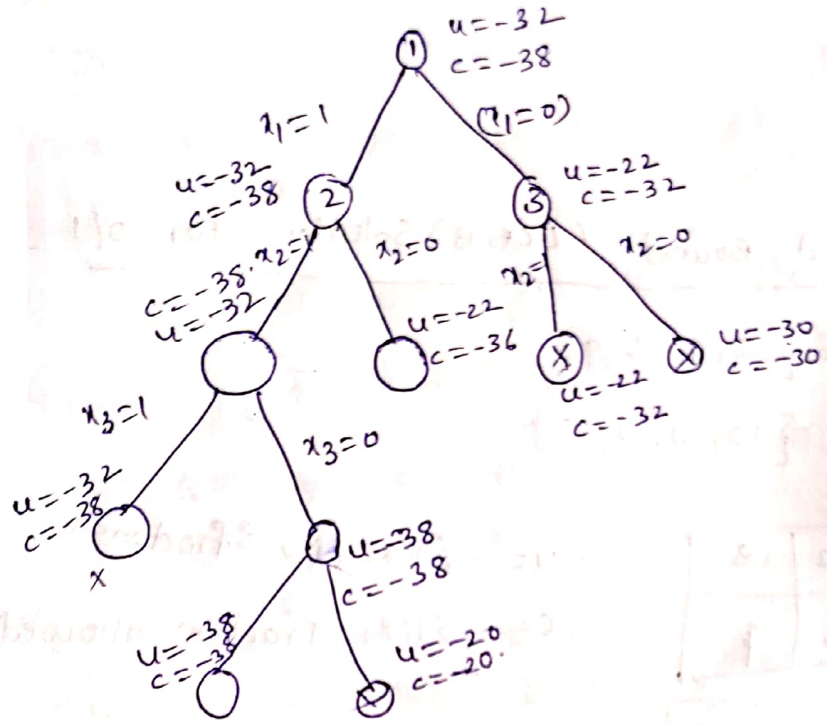
* [AS UB ↑, kill nodes]

18/3/19

FIFO Branch and Bound Sol?

p	10	10	12	18
w	2	4	6	9

$m = 15$
$n = 4$



20/3/19

Travelling Salesperson Problem using Branch and Bound -

Cost matrix -

	1	2	3	4	5	
1	∞	20	30	10	11	10
2	15	∞	16	4	2	2
3	3	5	∞	2	4	2
4	19	6	18	∞	3	3
5	16	4	7	16	∞	4
	↓					21

Row Reduction -

Reduce least values from each Rows.

	1	2	3	4	5	
1	∞	10	20	0	1	
2	13	∞	14	2	0	
3	1	3	∞	0	2	
4	16	3	15	∞	0	
5	12	0	3	12	∞	
	↓					
	0	0	3	0	0	= 4

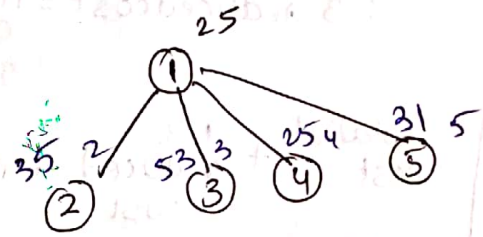
Column Reduction -

Deduce the present values from each column with least value in each column.

	1	2	3	4	5
1	∞	10	17	0	1
2	12	∞	11	2	0
3	0	3	∞	0	2
4	15	3	12	∞	0
5	11	0	0	12	∞

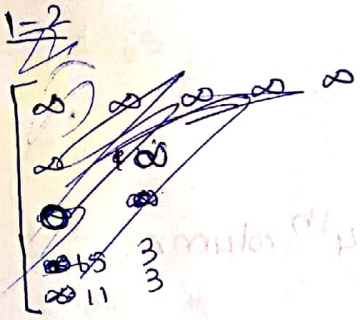
Total Reduction = Row Reduction + column Reduction
= 21 + 4 = 25

→



check

1-2



∞	∞	∞	∞	∞
∞	∞	11	2	0
0	∞	∞	0	2
15	∞	12	∞	0
11	∞	0	12	∞

Keep '∞' in 1st row and 2nd column. and also ∞ in '2-1'

Total reduction cost + 1-2 reduced cost + m(1,2)

= 25 + 0 + 10 = 35

1-2 reduced cost = 0.

→ check 0 in each row and column, if not there do row or column reduction.

1-3

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & 2 & 0 \\ 0 & 3 & \infty & 0 & 2 \\ 15 & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & 12 & \infty \end{bmatrix} \begin{matrix} -0 \\ -0 \\ -0 \\ -0 \\ -0 \end{matrix}$$

make (3,1) as ∞

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & 2 & 0 \\ \infty & 3 & \infty & 0 & 2 \\ 15 & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & 12 & \infty \\ \text{no-zero}(x) & 0 & 0 & 0 & 0 \end{bmatrix} \begin{matrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{matrix}$$

→ Reduce 11 from colum

(11)

1-3 reducedcost = 11

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & 11 & 2 & 0 \\ \infty & 3 & \infty & 0 & 2 \\ 4 & 3 & 12 & \infty & 0 \\ 0 & 3 & 0 & 12 & \infty \end{bmatrix}$$

Total reduced cost + 1-3 reduced cost + m(1,3)

25 + 11 + 17 = 53

1-4

Make (4,1) = ∞ and 1st Row & 4th column.

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

Total reduction cost + 1-4 reduced cost + m(1,4) = 25 + 0 + 0 = 25

1-5

make $(5,1) = \infty$ and 1^{st} row and 5^{th} column as ∞.

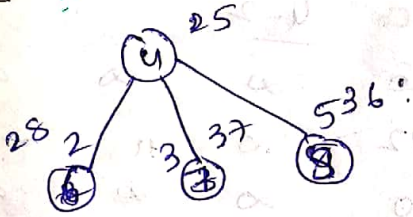
$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & 2 & \infty \\ 0 & 3 & \infty & 0 & \infty \\ 15 & 3 & 12 & \infty & \infty \\ \infty & 0 & 0 & 12 & \infty \\ 0 & 0 & 0 & 0 & \infty \end{bmatrix} \begin{matrix} 0 \\ 2 \\ 0 \\ 3 \\ 0 \end{matrix}$$

Reduction cost = 5

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

$25 + 5 + 1 = 31$.

2nd step -



1-4

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

4-2

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & 0 & \infty & \infty \\ 0 & 0 & 0 & \infty & 0 \end{bmatrix} \begin{matrix} 0 \\ 0 \\ \infty \\ 0 \end{matrix}$$

$\Rightarrow$ $25 + 0 + 3 = 28$

4-3

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

$25 + 0 + 12 = 37$

4-5 -

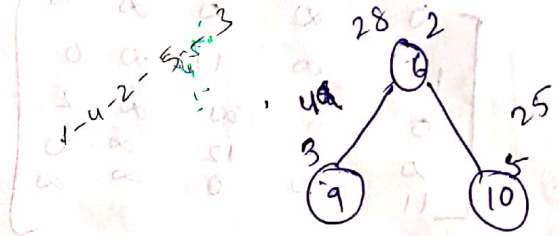
∞	∞	∞	∞	∞	∞
12	∞	11	∞	∞	11
0	3	∞	∞	∞	0
∞	∞	∞	∞	∞	∞
11	0	0	∞	∞	0
0	0	0	∞	∞	

Row Reduction cost = 11

∞	∞	∞	∞	∞
1	∞	0	∞	∞
0	3	∞	∞	∞
∞	∞	∞	∞	∞
11	0	0	∞	∞

25 + 11 + 0 = 36

3rd step -



4-2

∞	∞	∞	∞	∞
12	∞	11	∞	0
0	∞	∞	∞	2
∞	∞	∞	∞	∞
11	∞	0	∞	∞

2-3 (3,2) = ∞

∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞
0	∞	∞	∞	2	0
∞	∞	∞	∞	∞	∞
11	∞	∞	∞	∞	11
0	∞	∞	∞	2	

Row Reduction + column Reduction = 11 + 2 = 13

∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞
0	∞	∞	∞	0	0
∞	∞	∞	∞	∞	∞
0	∞	∞	∞	∞	0
0	∞	∞	∞	0	

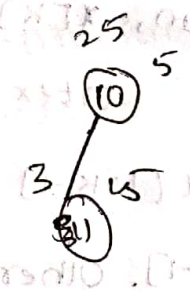
⇒ 25 + 13 + 11 = 49

2-5

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & 0 & \infty & \infty \\ 0 & \infty & 0 & \infty & \infty \end{bmatrix} \begin{matrix} \infty \\ \infty \\ 0 \\ \infty \\ 0 \end{matrix}$$

$25+0+0=25$

4th step -



2-5

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

5-3

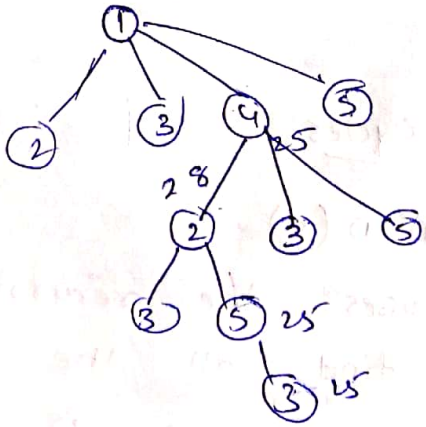
$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \end{bmatrix} \begin{matrix} \infty \\ \infty \\ 0 \\ \infty \\ \infty \end{matrix}$$

Row Reduction = 11

$\Rightarrow 25+0+0=25$

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \end{bmatrix} \begin{matrix} \infty \\ \infty \\ 0 \\ \infty \\ 0 \end{matrix}$$

25



[1-4-2-5-3]

Algorithm for Hamiltonian cycle -

Algorithm for generating a next vertex in Hc:

```
Algorithm NextValue(k)
//x[1:k-1] is a path of k-1 distinct vertices.
//If x[k]=0, then no vertex has as yet been
//assigned to x[k]. After execution, x[k] is assigne
// to the next highest numbered vertex which
//does not already appear in x[1:k-1] and is
// connected by an edge to x[k-1]. Otherwise x[k]=
// If k=n, then in addition x[k] is connected to x[
  {
  repeat
   {
   x[k]:= (x[k]+1) amod (n+1);
   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;
   if (j=k) then
   if ((k<n) or ((k=n) and G[x[n], x[1]] ≠0))
   then return;
    }
    } until( false);
    }
```

Finding Hamiltonian cycles -

```
Algorithm Hamiltonian(k)
// This algorithm uses the recursive formulation
of backtracking to find all the Hamiltonian
cycles of a graph. The graph is stored as an
```

//adjacency matrix G[1:n, 1:n]. All cycles begen at //node 1.

```
{
repeat
 {
NextValue(K);
if(x[K] = 0) then return;
if (K=n) then write (x[1:n])
 else Hamiltonian (K+1);
 } until (false);
  }
```

25/3/19

UNIT-V

LOWER BOUND THEORY

Comparision Trees -

The use of comparision trees for deriving Lower Bounds that are collectively called sorting and searching.

→ These trees are especially useful for modelling the way in which a large number of sorting and searching algorithm ~~works~~ box. work.

→ By appealing some elementary facts about trees, the lower Bounds are obtained.

→ Given a set 's' of distinct values on which an ordering relation "<" holds.

The sorting problem calls for determining a permutation of the integers 1 to n, say P(1) to P(n), such that the n-distinct values from 's' sorted in A[1:n] satisfy.

$A[P(1)] < A[P(2)] < \ldots\ldots < A[P(n)]$.

Sorting -

Consider the case in which there are 'n' numbers A[1:n] to be sorted are distinct. Any comparision blw A[i] and A[j] must result in one of two possibilities.

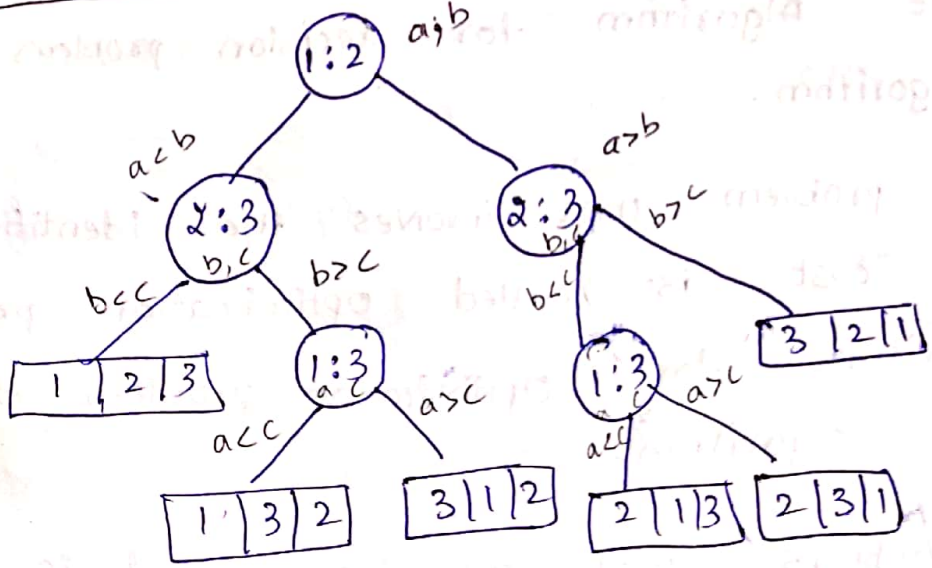
either $A[i] < A[j]$ or $A[j] > A[j]$. So, the comparision Tree is a Binary Tree. in which each internal node is labelled by

the pair i:j which represents the comparision of A[i] with A[j]. If A[i] is less than A[j], then the algorithm proceeds down the left Branch of the tree otherwise it proceeds down the right Branch.

→ The external nodes represent the termination of the algorithm. Associated with every path from the root to an external node is unique permutation. To see that this permutation is unique, note that the algorithm's allow or only permitted to move data and make comparisions.

→ The data movement on any path from root to an external node is the same no matter what is initial inputs are.

A comparision tree for sorting three elements-



asc order

26/3/19

NP hard and NP complete problems -

The classes P and NP -

There are two groups in which a problem can be classified. The first group consists of the problem that can be solved in polynomial time. (P)

Ex- Searching of an element from the list. O(logn), Sorting of element - O(n)

→ The second group consist of the problems that can be solved in non-deterministic polynomial (NP) time.

Ex- Knapsack problem - $O(2^{n/2})$, travelling sales person problem - $O(n^2 2^n)$.

→ Any problem for which answer is either yes or no is called as a decision problem. The Algorithm for decision problem is decision algorithm.

→ Any problem that involves the identification of optimal cost is called optimization problem. The algorithm for optimization problem is Optimization algorithm.

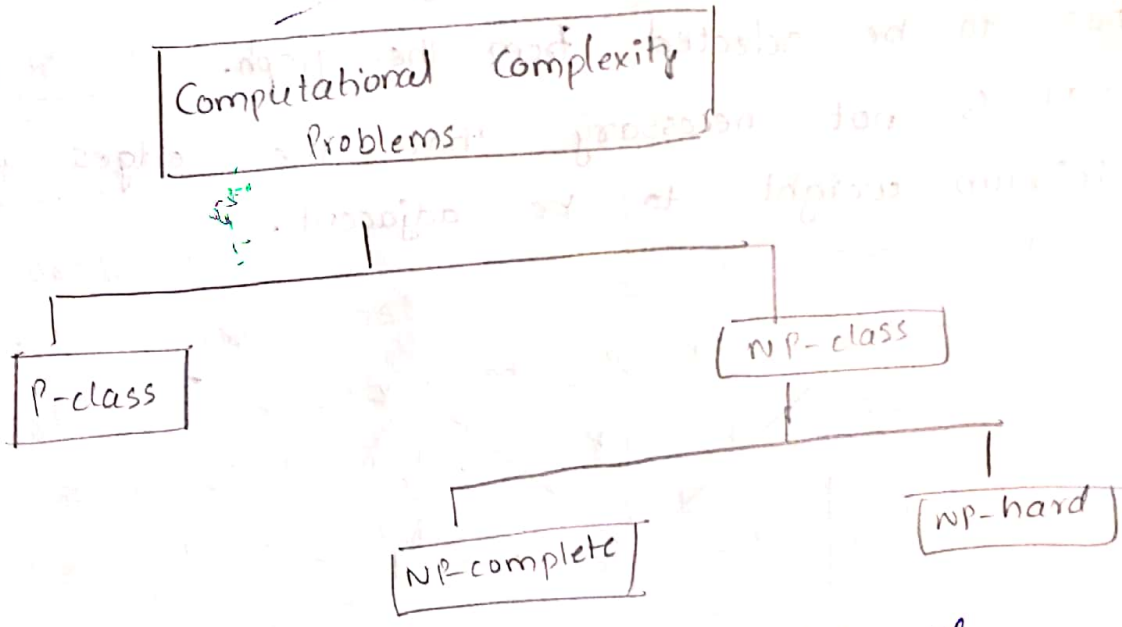
→ The problems that can be solved in polynomial time is called Class P or P problem

Ex- Searching of key elements, Sorting of elements.

→ The problems that can be solved in non-deterministic polynomial time is called class NP or NP problems.

Ex- Travelling sales person problem, fraph colouring problem, Knapsack problems.

→ The NP class problems further can be categorized into NP complete and NP hard problems.



→ A problem D is called NP complete if

① It belongs to class NP. and can also be Solved in polynomial time.

② If an NP hard problem can be solved in polynomial time then all NP complete problems can also be solved in polynomial time.

③ All NP complete problems are NP hard but all NP hard cannot be NP complete problems.

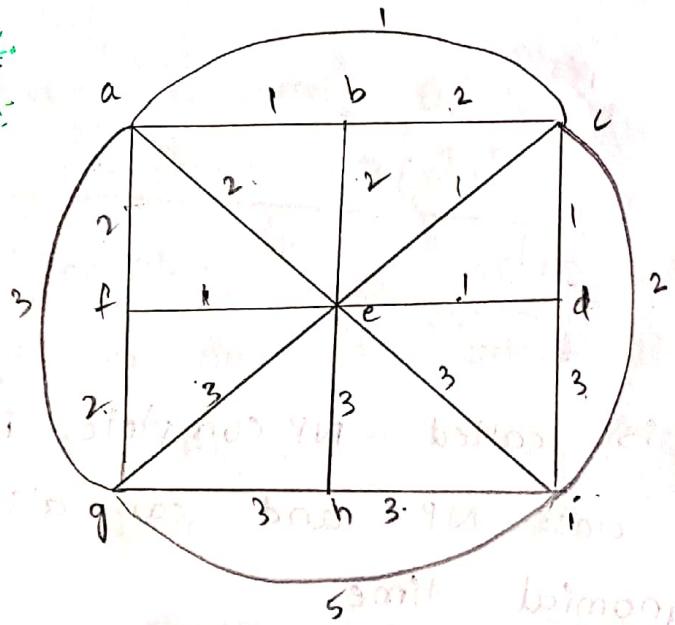
④ The NP-class problems are the decision problems that can be solved by non-deterministic polynomial algorithms.

Ex of P class problem-

Kruskal's Algorithm-

In Kruskal's Algorithm the minimum weight is obtained and also circuit should not be formed. Each time the edge of minimum weight has to be selected from the graph.

→ It is not necessary that the edges with minimum weight to be adjacent.

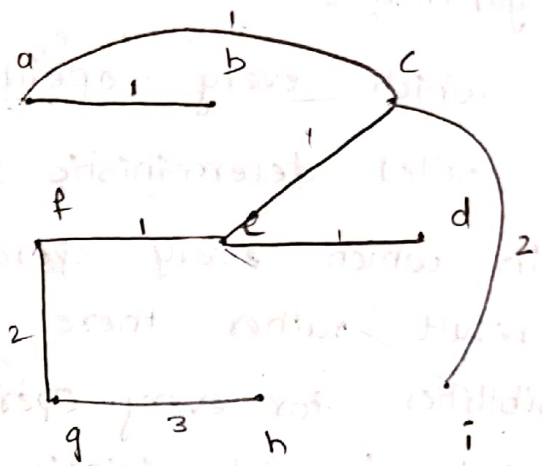


Edges.

a-b=1
b-c=1
f-e=1
e-e=1
e-d=1
c-d=1

a-e=2
a-f=2
b-c=2
b-e=2
f-g=2
e-i=2

a-g=3
g-h=3
h-i=3
e-i=3
d-i=3
g-e=3
e-h=3



cost = 1+1+1+1+1+2+2+3 = 12

28/3/19

Example of NP- class problem -

Travelling sales person problem -

This problem can be stated as given set of cities and cost to travel blw each pair of cities, determine whether there is a path that visit every city once and returns to the first city such that the cost travelled is less.

→ This problem is NP- Problem as there may exist some path with shortest distance blw the cities. If the soln is acheived by applying certain algorithm then travelling sales person problm is a NP complete problem.

If the soln is not acheived at all by applying an algorithm then the travelling sales person problem belongs to NP hard class.

Non- Deterministic Algorithm -

① The algorithm in which every operation is uniquely defined is called deterministic algorithm.

② The algorithm in which every operation may not have unique result rather there can be specified set of possibilities for every operation. Such algorithm is called Non- Deterministic Algorithm.

Non-Deterministic means no particular rule is followed to make the guess.

③ The non-deterministic algorithm is a two stage algorithm.

① Non-deterministic (guessing) stage -

Generate an arbitary string that can be thought of as a candidate soln.

② Deterministic (verification) stage -

In this stage it takes as input the candidate soln and instance to the problem and returns yes if the candidate soln represents actual soln.

```
Algorithm   Non- Determin( )
// A[1:n] is a set of elements have to determine
// the index i of A at which element x is
// located.
{
// The following for loop is the guessing stage
    for i:=1 to n do
    A[i] := choose(1);
```

// Next is the verification (deterministic) stage

```
if (A[i] = x) then
{
write (i);
success ();
}
write(0);
fail();
}
```

→ In the above given non-deterministic algorithm there are 3 functions used

① choose() - Arbitarily chooses one of the element from given input set.

② fail() - Indicates the unsuccessful completion.

③ success() - Indicates the successful completion.

29/3/19

Proving NP problems -

① Hamiltonian cycle -

This is a problem in which graph 'G' is accepted as input and it is asked to find a simple cycle in 'G' that visits each vertex of G exactly once and returns to its starting vertex. Such a cycle is called Hamiltonian cycle.

Theorem - Hamiltonian cycle is in NP.

Proof - Let 'A' be some non-deterministic algorithm to which graph 'G' is given as

input. The vertices of graph are numbered from 1 to N. call the algorithm recursively in order to get the sequence 'S'. This Sequence will have all the vertices without getting repeated. The vertex from which the sequence starts must be ended at the end. This check on the sequence 'S' must be in polynomial time 'n'. If there is a hamiltonian cycle in the graph 'G' then algorithm gives output 'YES'.

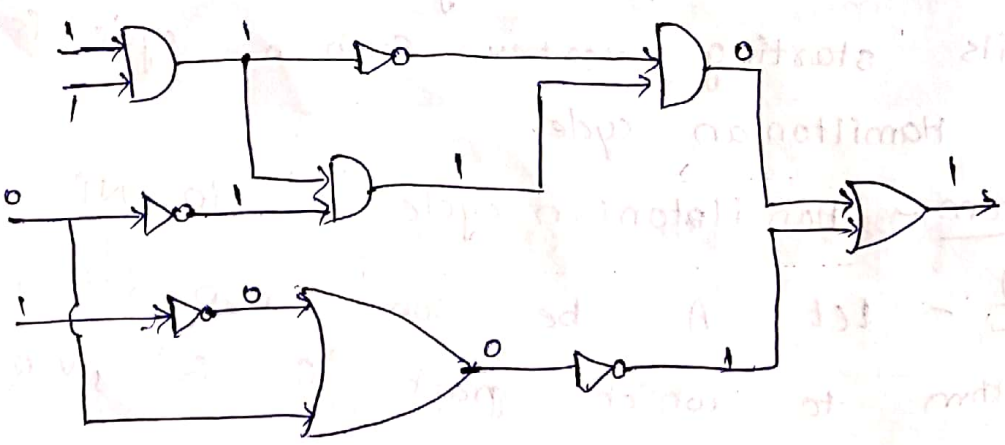
That means 'A' non-deterministically accepts the language hamiltonian cycle. It is therefore proved that hamiltonian cycle is in NP.

Circuit SAT -

This is a problem in which a boolean circuit is taken as input with single output node. and then finds whether there is an assignment of values to the circuits input so that it's output value as 1. This assignment of values is called satisfying assignment.

Theorem - CIRCUIT SAT is in NP

Proof -



→ construct a non-deterministic algorithm which works in polynomial time. Then it should have choose() method which can guess the values of input node as well as the output, then the algorithm says 'NO' if it gets the output as of boolean circuit 0 and similarly the algorithm says 'YES' if the output of the boolean circuit is 1. Thus, can say that circuit SAT in 'NP'.

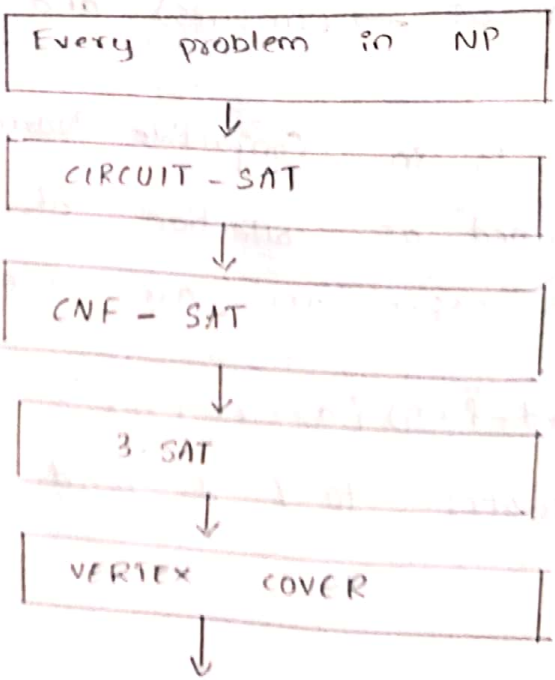
NP complete problems -

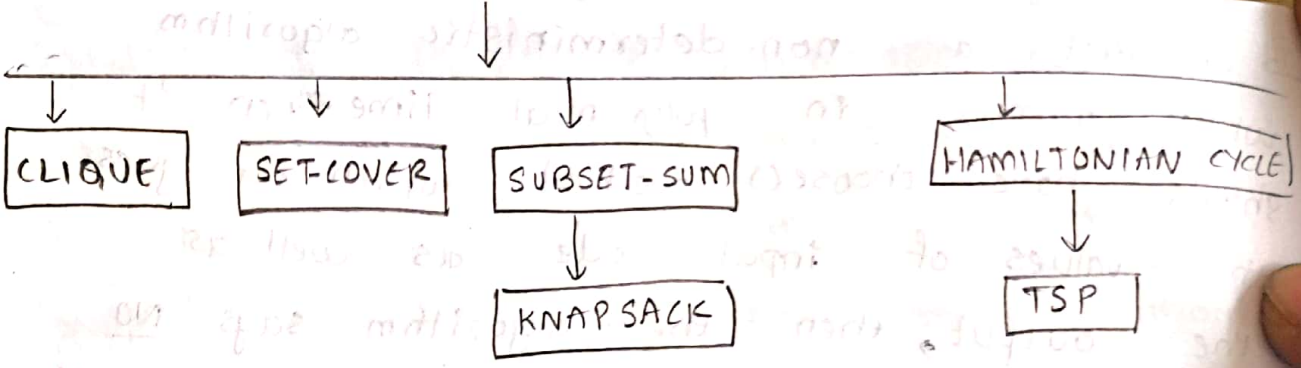
→ To prove whether a particular problem is in NP complete or not use polynomial time reducibility ie,

A $\xrightarrow{\text{Poly}}$ B and B $\xrightarrow{\text{poly}}$ C then A $\xrightarrow{\text{poly}}$

The reduction is an important task in NP completeness proofs.

→ The reductions can be illustrated as :





→ Various types of reductions are:

① Local replacement: In this reduction A→B by dividing input to A in the form of components and then these components can be converted to components of B.

② component design - In this reduction A→B by building special components for input B that enforce properties required by A.

30/3/19

The satisfiability problem -

① CNF-SAT problem - This problem is based on the Boolean formula. The Boolean formula has various Boolean operations such as OR, AND, NOT. There are some notations such as → (implies) and ↔ (if and only if).

→ A Boolean formula is in Conjuctive Normal form (CNF) if it is formed as collection of sub-expressions. These sub expressions are called clauses.

Ex - $(\bar{a}+b+c+\bar{d})(\bar{b}+d+\bar{f}+h)(a+c+e+\bar{h})$.

This formula evaluates to 1 if b, c, d are 1.

→ The CNF SAT is a problem which takes Boolean formula in CNF form and checks whether any assignment is there to Boolean values so that formula evaluates to 1.

Theorem - CNF-SAT is in NP complete.

Proof - let 's' be the Boolean-formula for which, can construct a simple non-deterministic algorithm which can guess the values of variables in Boolean formula and then evaluates each clause of S. If all the clauses evaluate 's' to "1" then 's' is satisfied.

Thus, CNF-SAT is in NP-complete.

② A 3 SAT problem -

A 3 SAT problem is in which takes a Boolean formula 's' in CNF form with each clause having exactly three literals and check whether 's' is satisfied or not.

Ex - $(\bar{a}+b+\bar{g})\ (c+\bar{e}+f)\ (\bar{b}+d+\bar{f})\ (a+e+\bar{h})$

Theorem - 3 SAT is in NP complete.

Proof - let 's' be the Boolean formula having 3 literals in each clause for which it can construct a simple non deterministic algorithm which can guess an assignment of Boolean values to S. If the S - is evaluated as '1' then S - is satisfied. Thus, 3 SAT is in NP complete.

Other NP complete problems -

① The 0/1 knapsack problem - It can be prooved as NP complete by reduction form SUM OF SUBSET problem.

② Hamiltonian cycle - It can be proved as NP complete, by reduction from vertex cover.

③ Travelling Sales person Problem - It can be proved as NP complete, by reduction from Hamiltonian cycle.

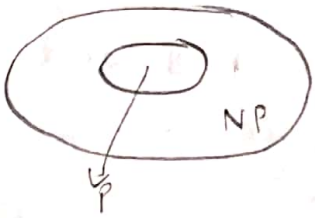
8/4/19

Properties of NP- complete and NP-Hard problems -

→ As 'P' denotes the class of all deterministic polynomial language problems and NP denotes the class of all non-deterministic polynomial language problems. Hence $P \subseteq NP$.

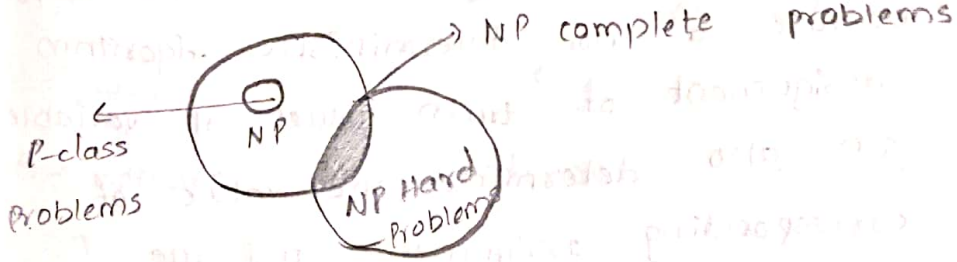
→ The question of whether $P = NP$ or not? Holds the most famous outstanding problem in the computer science.

→ Problems which are known to lie in P are often called as tractable problems which lie outside of P are often termed as intractable. Thus the question of whether $P = NP$ or $P \neq NP$ is the same as that of asking whether there exists problems in NP which are in tractable or not. The relationship between P & NP problems is



→ Let A and B are 2 problems then problem A reduces to B iff there is a way to solve A by deterministic polynomial time algorithm using a deterministic algorithm that solves B in polynomial time A reduces to B can be denoted as A ∝ B.

→ A NP problem such that if it is in P then NP=P. If a problem has this property then it is called NP Hard. Thus the class of NP complete problem is the intersection of NP & NP Hard classes.



Relationship blw P, NP, NP- complete and NP- Hard problems.

→ Normally the decision problems are NP complete but optimization problems are NP Hard.

→ There are some NP- Hard problems that are not NP complete.

Two problems P and Q are said to be polynomially equivalent iff $P \propto Q$ and $Q \propto P$.

Cook's Theorem -

Proof of Cook's Theorem -

Scientist stephen Cook stated that Boolean satisfiability problem is NP complete.

Proof -

Any instance of Boolean satisfiability problem is a boolean expression in which boolean variables are combined using Boolean Operators. An Expression is satisfiable if its value results to be true on some assignments of Boolean variables. The Boolean satisfiability problem is in NP. This is because a non-deterministic algorithm can guess an assignment of truth values of variables This algorithm can also determine the value of expression for corresponding assignment and can accept if entire expression is true.

The algorithm is composed of

① Input tape where in tape is divided infinite number of cells.

② The read/write head which reads each Symbol from tape.

③ Each cell contain only one Symbol at a time.

④ Computation is performed in number of states

⑤ The algorithm terminates when it reaches to accept state.

→ The conjuction clauses for Boolean Expression are given Below.

Clauses	Meaning	Time
T_{ijo}	cell i of input tape contains Symbol j	$O(P(n))$
Q_{so}	Initial state	$O(1)$
H_{oo}	Initial position of tape head	$O(1)$
$T_{ijK} = T_{ij(K+1)} \vee H_{jK}$	Tape remains unchanged unless written.	$O(P(n^2))$
$Q_{qK} \rightarrow \sim Q_{qK}$	one state at a time	$O(P(n))$
$H_{iK} \rightarrow \sim H_{iK}$	One read/write head position at a time	$O(P(n^2))$
$T_{ijK} \rightarrow \sim T_{ijK}$	One symbol pertape cell at a time	$O(P(n^2))$

H- denotes Head, Q- denotes states and T- denotes tape

The disjunction clause for this algorithm can be written as.

$(H_{iK} \wedge Q_{qK} \wedge T_{ijK}) \rightarrow (H_{(i+j)(K+1)} \wedge Q_{q(K+1)} \wedge T_{ij(K+1)})$	Possible Transitions	$O(P(n^2))$
Disjunction of all clauses of	moving to accept state	$O(1)$

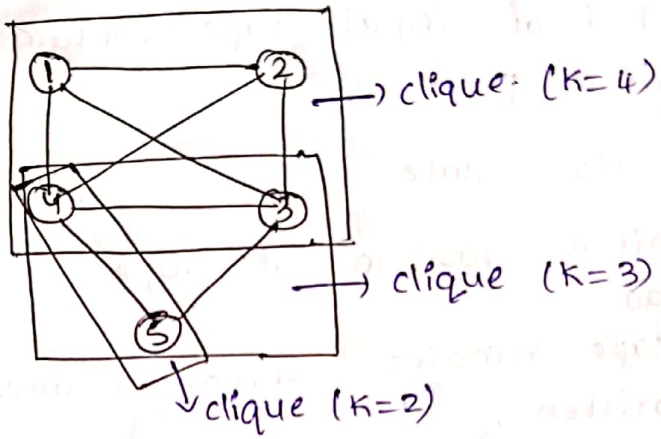
→ If B is a satisfiable, then there is an accepting state in the algorithm.

9/4/19

Clique Decision Problem (CDP)-

Clique is defined as any sub graph of a graph is a complete graph then that subgraph is called as clique.

Ex -



CDP-

Finding out whether there is a clique in the graph or not is called as clique Decision Problem.

Clique Optimization problem -

Finding out the maximum number of a clique in a graph is called as clique optimization problem.

Theorem -

CNF - Satisfiability $\propto$ Clique decision problem.

(prove CDP is NP hard)

Proof- let $F = \wedge_{1 \le i \le k} C_i$ be a propositional formula in CNF. let $x_i, 1 \le i \le n$, be the variables in F. Show how to construct from F a graph $G = (V, E)$ such that G has a clique of size atleast k if and only if F is satisfiable.

If the length of F is m, then G is obtained from F in $O(m)$ time. Hence, if it have a polynomial time algorithm for CDP, then it can obtain a polynomial time algorithm for CNF-satisfiability using this construction.

For any F, $G = (V, E)$ is defined as follows.

$V = \{ \langle \alpha, i \rangle / \alpha$ - is a literal in clause $C_i \}$ and

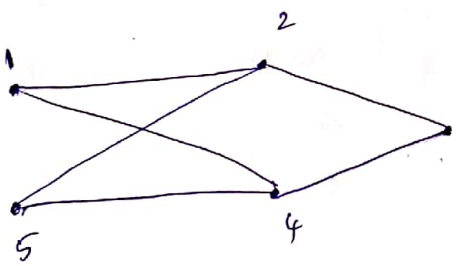
$E = \{ (\langle \alpha, i \rangle, \langle \beta, j \rangle) / i \neq j$, and $\alpha \neq \beta \}$

12/4/19

Node Cover Decision problem -

A set $S \subseteq V$ is a node cover for a graph $G = (V, E)$ if and only if all edges in E are incident to atleast one vertex in S. The size $|S|$ of the cover is the no. of vertices in S.

Ex -



$S = \{2, 4\}$ is a node cover of size 2.

$S = \{1, 3, 5\}$ is a node cover of size 3.

Theorem - The clique decision problem & The node cover decision problem.

Proof - let $G = (V, E)$ and K define an instance of CDP. Assume that $|V| = n$, construct a graph G' such that G' has a node cover of size atmost $(n - K)$ iff G has a clique of size atleast K.

Graph G' is given by $G' = (V, \bar{E})$, where $\bar{E} = \{(u, v) / u \in V, v \in V$ and $(u, v) \notin E\}$. The set G' is known as the complement of G.

→ Now, show that G has a clique of size at least K iff G' has a node cover of size atmost $n-K$. let K – be any clique in G. since, there are no edges in $\bar{E}$ connecting vertices in K, the remaining $n-|K|$ vertices in G' must cover all edges in $\bar{E}$.

→ Similarly, if S – is a node cover of G', then $V-S$ must form a complete subgraph in G.

Since, G' can be obtained from G in polynomial time, CDP can be solved in polynomial deterministic time if it have a polynomial time deterministic algorithm for NCDP.

13/4/19 **Time Complexity** - (Step count method)

s/e = steps per execution

(1)

Statement	s/e	Frequency	total steps
Algorithm sum (a, n)	0	—	0
{	0	—	0
s := 0+0;	1	1	1
for i := 1 to n do	1	n+1	n+1
s := s + a[i];	n+1	n	n
return s;	n	1	1
}	1	1	1
	0	—	0

Total = $1 + (n+1) + n + 1 = 2n + 3$.

(2)

Statement	s/e	Frequency	total steps.
Algorithm add (a, b, c, m, n)	0	—	0
{	0	—	0
for i := 1 to m do	m+1	m+1	m+1
For j := 1 to n do	m(n+1)	m(n+1)	n(m+1)
c[i,j] = a[i,j] + b[i,j];	mn	mn	mn
}	0	0	0

Total = $(m+1) + m(n+1) + mn$

$= m+1+mn+m+mn$

$= 2mn + 2m + 1$

$= 2m(n+1) + 1$