UNIT-IV: BACKTRACKING, BRANCH AND BOUNDS

Backtracking

General Method:

Let $(a_1, a_2, a_3, \ldots\ldots, a_n)$ be a set of 'n' tupples.

Now, first construct a partial solution vector from the given set.

$\{a_1, \ldots\ldots, a_{1k}, a_{1k+1}\}$ => Partial Solution Vector

First, take on tupple, and then add other, then check for the constraints of the problem, if that satisfies, add the other tupple, else, go to first tupple (backtracking) and change it.

Problems in Backtracking:

i) 8- Queens Problems
ii) Graph Colouring
iii) Hamilton Cycles
iv) 0/1 knapsack

8- Queens Problem:

→ 4 Queens

Solution Vector:

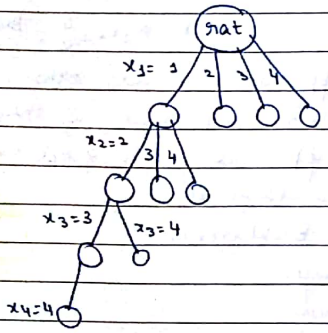
x_1	x_2	x_3	x_4
1	3	2	
2	4	1	3

	Q_1		
			Q_2
Q_3			
		Q_4	

Problem constraint: No two queens should be attacked

There are two constraints in this problem:

- Explicit Constraint ⇒ $x_i \le n$ and $x_i = 0$ or 1
- Implicit Constraint ⇒ To get some solution vector from solution sample



⇒ Solution Sample

Out of the solution sample, we have to select the solution vector which satisfies all the constraints of the problem.

General Method of Backtracking (Recursive Algorithm):

```
Algorithm BackTrack(k)
{ for each x[k] ∈ T (x[1], x[2], ....., x[k-1]) do
  { if (B_k (x[i], ....., x[k] ≠ 0) then
    {
      if (x[i], ..... x[k]) is a path to answer node
      then write x[i-k]
      if [k<n) then BackTrack(k+1)
    }
  }
}
```

8-Queens Problem:

Constraints:

i) $|i-j| = |k-l|$
→ means diagonals
$|i+j| \neq |k+l|$

ii) If (i,j) and (k,l) are 2 queens positions, then
$i \neq k$ and $j \neq l$

Here: (i,j) = (row, column), $(k-l)$ = (row, column)

(1,1)			
	(2,2)		
		(3,3)	
			(4,4)

⇒ Diagonals

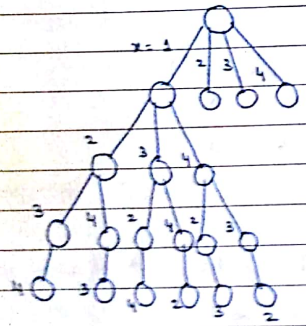
Algorithm of 8-Queens:

```
Bool Algorithm place(k,i)
{
  for j:=1 to k-1 do
    if (x[j]=i)) // two in the same column
      or
    Abs(x[j]-i) = Abs(j-k) // same diagonal
      then return false;
  return true;
}
```

Algorithm of N-Queens

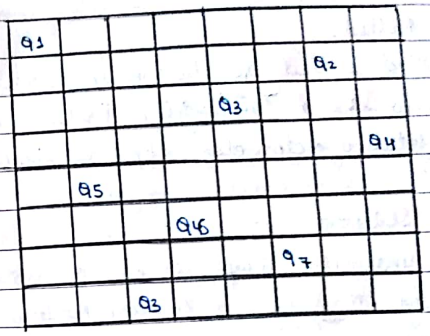
```
Algorithm NQueens (k, n)
{
  for i := 1 to n do
    if (place (k, i)) then
    {
      x[k] := i;
      if (k=n) then write (x[1, ..... , n])
      else NQueens (k+1, N)
    }
}
```

Possible Solutions to Select Solution Vector for 4-Queens



Total number of possible solutions = 24

8-Queens Solution:



Partial Solution Vector:

{x1	x2	x3	x4	x5	x6	x7	x8 }
1	7	5	8	2	4	6	3

P.T.O →

Graph Colouring

General Method:

Construct partial solution vector $x_1, x_2, x_3, \ldots, x_k$. If the new element x_{k+1} satisfies problem constraints, add it to solution vector, else, apply backtracking.

Problem Statement:

Given undirected graph 'G' has 'n' vertices, we must assign 'm' number of colours (minimum no. of colours) to that no two adjacent vertices have same colour.

x_1 x_2 x_3 ..... x_n

0 0 0 0 => Step 1: assign no colour to all vertices

1 => Step 2: adjacent colour to node 1

1 2 => Step 3: Assign colour 1 to node 2 if node 1 is adjacent to node 2 and node 2 has the same colour as colour of node 1, assign colour 2 to node 2

Continue till x_{k+1}

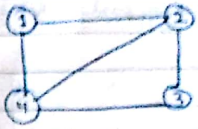
Construct Binary Adjacency Matrix for given graph 'G'.

G(i,j) = 1 => i and j are adjacent nodes

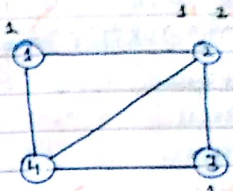
G(i,j) = 0 => i and j are not adjacent nodes

→

eg



=>



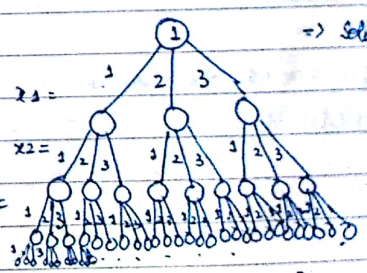
x_1	x_2	x_3	x_4
0	0	0	0
1	X	1	X
	2		X
			3

→ x_1 x_2 x_3 x_4 = 1 2 1 3

State Space / Solution Space for the graph

= 4 levels etc

= x_1 x_2 x_3 x_4



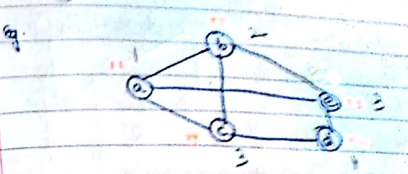
=> Solution Space

∴ Total no. of possibilities = $n^m = 4^3$, (no. of vertices) (no. of leaf nodes)

```
Algorithm mcoloring(k) // Assume partial solution is generated upto k-1
{
  repeat
  {
    nextcolor(k) // return the colour assigned to a predecessor node
    if (x[k] = 0) then return;
    if (k=n) write (x[1] ..... x[n])
    else mcoloring (k+1);
  } until (false)
}

Algorithm nextcolor (k)
{
  repeat
  {
    x[k] := (x[k]+1) mod (m+1); // initially each vertex is assigned no colour
    if (x[k]=0) then return;
    for j := 1 to n do
    {
      (adjacency condition) (same colour condition)
      if (G(k,j) ≠ 0 and x[k] = x[j])
      break;
    }
    if (j = n+1) then return;
  } until (false)
}
```

Eg:



x1	x2	x3	x4	x5
0	0	0	0	0

nextcolor(x1)

x[x1] := (0+1) mod 5 = 1;

for j = 1 to 5

G(x1,x1) ≠ 0 — x1 x1 — j = 1

G(x1,x2) ≠ 0 — 1 ≠ 0 — j = 2

G(x1,x3) ≠ 0 — x1 x3, 1 = 0 — j = 3

G(x1,x4) ≠ 0 — x1 x4, 1 = 0 — j = 4

G(x1,x5) ≠ 0 — x1 x5, 1 = 0 — j = 5

j = 6 = n+1 = 5+1 = 6

break;

x[x] ≠ 0

x = 1 ≠ 5

mcoloring(2)

nextcolor(x2)

x[x2] := (1+1) mod (4+1) = 2

j = 1 to 5

G(x2,x1) ≠ 0 — x=1 — j = 1

j = 1 = 5+1 ⇒ false

x[x2] := (2+1) mod 4 = 3

j = 1 to 5

G(x2,x1) ≠ 0 — x=1 — j = 1

G(x2,x2) ≠ 0 — j = 2

G(x2,x3) ≠ 0 — x=3 — j = 3

G(x2,x4) ≠ 0 — x=3 — j = 4

G(x2,x5) ≠ 0 — x=3 — j = 5

j = 6 = n+1 = 6

return

x[x] = 0 → false

k = n → false

mcoloring (3)

03-'18

Hamiltonian Cycle

Problem Statement:

Given Graph 'G' travels all edges exactly once and reaches to starting point. The path contains $(n+1)$ vertices, v_1 and v_{n+1} are the same. Start at vertex 1, visit all other ~~edges~~ vertices by travelling all edges exactly once and reach back to starting point.

x_1	x_2	x_3	.....	x_n	x_{n+1}	
1	0	0	.....	0	1	→ initial solution vector

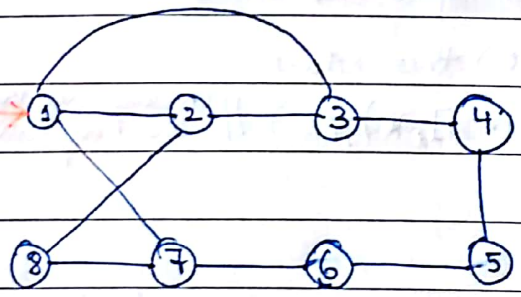
To select new vertex, k^{th} vertex, we have to verify 2 conditions:

i) x_{k-1} and x_k, both vertices are adjacent

ii) Path is generated upto $k-1^{th}$ vertex

To select last vertex, we must verify if a path exists from x_n to x_{n-1}

Eg.



2 paths:

(i) 1, 3, 4, 5, 6, 7, 8, 2, 1

(ii) 1, 2, 8, 7, 6, 5, 4, 3, 1

NOTE: It is like travelling salesman problem, but we consider different weights in travelling salesman problem, but in this problem, all the vertices have same distance, i.e., all the edges have same values.

Binary Adjacency Matrix

$G[i,j] = 1 \Rightarrow i, j$ are adjacent $G[i,j] = 0 \Rightarrow i, j$ are not adjacent

```
Algorithm Hamiltonian (k)
{
  repeat
  {
    nextVertex (k); //returns highest index adjacent to i
    if (x[k] = 0 ) then return
    if (k = n) then write(x[1] ---- x[n]) // write solution
    else
      Hamiltonian (k+1)
  } until (false)
}
Algorithm nextVertex (k)
{
  repeat
  {
    x[k] = (x[k]+1) mod (n+1);
    if (x[k]=0 ) then return
    if ( G ( x[k-1],x[k] ≠0 ) // checking if vertices k-1 and k are adjacent
    then
    {
      for j := 1 to k-1 // to verify if the vertex x[k] is distinct vertex or already visited vertex
      if [x[j] = x[k]) then break; // vertex are visited already
      if (j=k)
      {
        if((k<n) or((k=n) and G(x[n], x[1]) ≠ 0))
          then return;
      }
    }
  }
}
```

Branch and Bound

General Method:

We find solution vector from solution space tree.

We have three types of branching functions to construct state space tree.

i) FIFO - First In First Out

ii) LIFO - Last In First Out

iii) LC-Search - Least Cause Function

We need to discuss about 3 nodes in solution space tree or to construct state space tree

i) Live Node (L node)
   - The node is generated but not explored.

ii) Explore Node (E node)
   - The node is generated and being explored currently.

iii) Dead Node
   - The node cannot generate solution vector.

Difference between Branch and Bound and Backtracking

Branch and Bound contains branching function, use it we generate solution state space tree, then using bounding function, we decide whether we can explore the tree or not, but no such functions exist in Backtracking Method.

While generating solution space tree, we generate a bounding function and use it to explore the current node.

Live Node
- Live Node is a node that has been generated but both children have not yet been generated.

Explore Node
- Explore Node is a live node whose children are currently being explored.

Dead Node
- Dead Node is a generated node that is not to be expanded or explored any further.

We have 3 problems in Branch and Bound
→ 15 Puzzle Problem
→ Travelling Salesman Problem [TVSP]
→ 0/1 Knapsack Problem

15 Puzzle Problem

Problem Statement
- We take one frame, the fram is divided into 16 slots, each slot is assigned one number, one position is empty, we perform some moves so that the final state / goal state is reached.

1	3	4	15
2		5	12
7	6	11	14
8	9	10	13

Initial State

→

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

Goal State

By using Brute Force, it takes 16! possibilities, which is very difficult to search.
So, we use Branch & Bound.
We apply LC-search bound function.
For each and every node, we must calculate
$c(x) = f(h(x)) + g(x)$
where x = node, f = function (non decreasing fn)
h(x) = distance from root node to node x
g(x) = Any extra effort we are putting to reach root node to node 'x'
c(x) = cost function
Position of i = slot numbers (in black) in given frame
For given initial state, we find whether it is possible to reach goal state or not by using the formula:

$$\sum_{i=3}^{16} less(i) + x$$

→ If the function returns even value, then the solution is possible
→ If the function returns odd value, then the solution is not possible
→ less(i) is no. of times j such that j is less than i but Position of i > Position of j

Eg.

1	3	2	

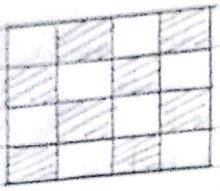
less(1) = 0 ⇒ no element after 1 is less than 1
less(3) = 1 ⇒ 1 element after 3 is less than 3
less(2) = 0 ⇒ no element after 2 is less than 2

Eg.

1	3	4	15
2		5	12
7	6	11	14
8	9	10	13

less(1) = 0 less(2) = 0 less(3) = 1
less(3) = 1 less(5) = 10 less(6) = 0
less(4) = 1 less(7) = 0 less(11) = 3
less(15) = 11 less(12) = 6 less(14) = 4

Consider empty cell value = 16
$\sum less(i) = 37$ $x = 0$ less(9) = 0 less(9) = 0 less(10) = 0
$\Rightarrow \sum less(i) + x = 37$ = odd ⇒ It is not possible less(13) = 0



Shaded region:
x value : 1
Non shaded region:
x value = 0

Eg:

1	2	3	4
5	6		8
9	10	7	11
13	14	15	12

less(1) = 0 less(5) = 0 less(9) = 1 less(13) = 0
less(2) = 0 less(6) = 0 less(10) = 1 less(14) = 0
less(3) = 0 less(7) = 9 less(11) = 0 less(15) = 0
less(4) = 0 less(8) = 1 less(12) = 0

$\sum_{i=1}^{16}$ less(i) = 15 x = shaded region = 1

$\therefore \sum_{i=1}^{16}$ less(i) + x = 15 + 1 = 16 = even ⇒ solution is possible

level 1

1	2	3	4
5	6		8
9	10	7	11
13	14	15	12

left right down up

level 2

1	2	3	4
5		6	8
9	10	7	11
13	14	15	12

1	2	3	4
5	6	8	
9	10	7	11
13	14	15	12

1	2	3	4
5	6	7	8
9	10		11
13	14	15	12

1	2		4
5	6	3	8
9	10	7	11
13	14	15	12

Calculate bounding function for each and every node

c(x) = f(h(x)) + g(x)

h(x) = difference in levels

g(x) = no. of numbers different from node in levels to goal state

Bound function for left = 1 + 5 = 6
Bound function for right = 1 + 5 = 6
Bound function for down = 1 + 3 = 4
Bound function for up = 1 + 5 = 6

Which node gives min cost = currently exploring node
Perform left, right, down of a min cost node,
exclude opposite operation (up)

level 2

1	2	3	4
5	6	7	8
9	10		11
13	14	15	12

left right ~~up~~ down

level 3

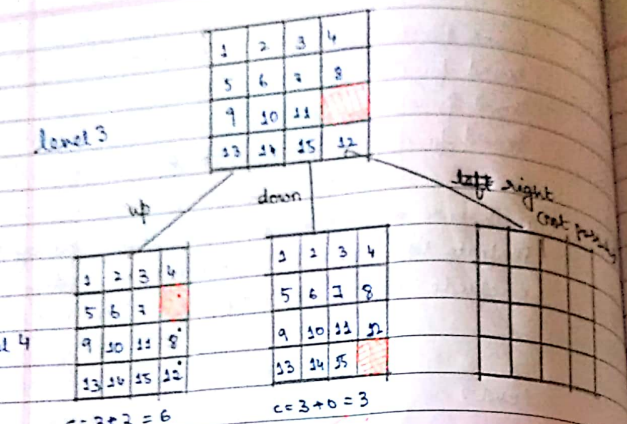
1	2	3	4
5	6	7	8
9		10	11
13	14	15	12

1	2	3	4
5	6	7	8
9	10	11	
13	14	15	12

1	2	3	4
5	6	7	8
9	10	15	11
13	14		12

Bound function for left = 2 + 4 = 6
Bound function for right = 2 + 2 = 4
Bound function for down = 2 + 4 = 6

Which node gives min cost = currently exploring node
Perform left, right, left, down of a min cost node, exclude opposite operation ~~down~~ (left)



down = final / goal state

∴ To reach goal state, we perform down, right, down operations on the initial state.

➡ Travelling Salesperson Problem [TVSP]

Problem Statement : We must covers all the vertices in the given graph (directed) with minimum cost, exactly once and reaches back to same original city.

G(V, E) => Directed Graph (Cities, Edges)

(1, π) => Starts at vertex 1, covers all other cities (n-1)! exactly once and returns back to vertex 1

Using Brute Force, Time Complexity = (n-1)! = n!

By using LC-Search, we find a solution for the problem

Solution has two steps

Step 1 : Find out reduced cost matrix

↳ a) Find Row Reduced Cost Matrix (R)
(identify minimum element from each row in cost matrix, subtract minimum element from each element in the row)

↳ b) Find Column Reduced Cost Matrix
(identify minimum element from each column in row reduced cost matrix, subtract minimum element from each element in the column)

↳ c) Find Cumulative reduced cost matrix
(row cost + column cost)

Step 2 : $\hat{c}(S) = c(R) + A(i,j) + r$

↳ a) (i, j) => Change all entries of row i in matrix A to infinite

↳ b) (i, j) => Change all entries of column j in matrix A to infinite

↳ c) (i, j) => Change (j, 1) entry to infinite

R = Root node A[] = Reduced cost matrix S

S = minimum cost child node, S be a child of R, not leaf node

c(S) = Cost of child node S c(R) = Reduced cost of matrix A

A(i, j) = A(R, S) r = Reduced cost of S

→

Eg. Given cost Matrix:

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

Row Reduced Cost Matrix

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

Row Reduced Cost ⇒ 21 (10+2+2+3+4)

Column Reduced Cost Matrix

	1	2	3	4	5
1	∞	10	17	0	1
2	12	∞	13	2	0
3	0	3	∞	0	2
4	15	3	12	∞	0
5	11	0	0	12	∞

1 0 3 0 0

⇒ Column Reduced Cost = 1+0+3+0+0 = 4

A =

∴ Cost (R): c(R) = rcost + ccost = 21 + 4 = 25

→

$\hat{c}(5) = c(R) + A(i,j) + r$

$\hat{c}(2) = c(R) + A(1,2) + r$

= 25 + 10 + 0 = 35

r:

	1	2	3	4	5	
1	∞	∞	∞	∞	∞	- 0
2	∞	∞	11	2	0	- 0
3	0	∞	∞	0	2	- 0
4	15	∞	12	∞	0	- 0
5	11	∞	0	12	∞	- 0
	0	0	0	0	0	

r = rcost + ccost = 0 + 0 = 0

$\hat{c}(5) = c(R) + A(i,j) + r$

$\hat{c}(3) = c(R) + A(1,3) + r$

= 25 + 17 + 11 = 53

r:

	1	2	3	4	5	
1	∞	∞	∞	∞	∞	- 0
2	12	∞	∞	2	0	- 0
3	∞	3	∞	0	2	- 0
4	15	3	∞	∞	0	- 0
5	11	0	∞	12	∞	- 0
	11	0	0	0	0	

r = rcost + ccost = 0 + 11 = 11

→

$\hat{c}(5) = c(R) + A(i,j) + r$

$\hat{c}(4) = c(R) + A(1,4) + r$

$= 25 + 0 + 0 = 25$

r:	1	2	3	4	5	
1	∞	∞	∞	∞	∞	-0
2	12	∞	11	∞	0	-0
3	0	3	∞	∞	2	-0
4	∞	3	12	∞	0	-0
5	11	0	0	∞	∞	-0
	0	0	0	0	0	

r: rcost + c cost = 0+0 = 0

$\hat{c}(5) = c(R) + A(i,j) + r$

$\hat{c}(5) = c(R) + A(1,5) + r$

$= 25 + 1 + 5 = 31$

r:	1	2	3	4	5	
1	∞	∞	∞	∞	∞	-0
2	12	∞	11	2	∞	-2
3	0	3	∞	0	∞	-0
4	15	3	12	∞	∞	-3
5	∞	0	0	12	∞	-0

Row Reduced Cost Matrix

	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	10	∞	9	0	∞
3	0	3	∞	0	∞
4	12	0	9	∞	∞
5	∞	0	0	12	∞
	0	0	0	0	0

Column Reduced Cost Matrix

	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	10	∞	9	0	∞
3	0	3	∞	0	∞
4	12	0	9	∞	∞
5	∞	0	0	12	∞

r = rcost + c cost = 5+0 = 5

Currently exploring node = minimum cost node = 1 to 4

A:

	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	12	∞	11	∞	0
3	0	3	∞	∞	2
4	∞	3	12	∞	0
5	11	0	0	∞	∞

$\hat{c}(2) = c(R) + A(4,2) + r$

$= 25 + 3 + 0 = 28$

$\hat{c}(3) = c(R) + A(4,3) + r$

$= 25 + 12 + 13 = 50$

r:	1	2	3	4	5	
1	∞	∞	∞	∞	∞	-0
2	∞	∞	11	∞	0	-0
3	0	∞	∞	∞	2	-0
4	∞	∞	∞	∞	∞	-0
5	11	∞	0	∞	∞	-0
	0	0	0	0	0	

r:	1	2	3	4	5	
1	∞	∞	∞	∞	∞	-0
2	12	∞	∞	∞	0	-0
3	∞	3	∞	∞	2	-0
4	∞	∞	∞	∞	∞	-0
5	11	0	∞	∞	∞	-0
	11	0	0	0	0	

$\hat{c}(5) = c(R) + A(4,5) + r = 25 + 0 + 11 = 36$

r:	1	2	3	4	5	
1	∞	∞	∞	∞	∞	-0
2	12	∞	11	∞	∞	-11
3	0	3	∞	∞	∞	-0
4	∞	∞	∞	∞	∞	-0
5	11	0	0	∞	∞	-0
	0	0	0	0	0	

∴ Currently exploring node = minimum cost node = 4 to 2

A:	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	12	∞	11	∞	0
3	0	∞	∞	∞	2
4	∞	∞	∞	∞	∞
5	11	∞	0	∞	∞

$\hat{c}(2) = c(R) + A(2,3) + 1$

$= 28 + 0 + 11 = 52$

$\hat{c}(2) = c(R) + A(2,3) + 1$

$= 28 + 0 + 0 = 28$

A:

	1	2	3	4	5	
1	∞	∞	∞	∞	∞	-0
2	∞	∞	∞	∞	∞	-0
3	∞	∞	∞	∞	2	-2
4	∞	∞	∞	∞	∞	-0
5	11	∞	∞	∞	∞	-11
	0	0	0	0	0	

A:

	1	2	3	4	5	
1	∞	∞	∞	∞	∞	-0
2	∞	∞	∞	∞	∞	-0
3	0	∞	∞	∞	∞	-0
4	∞	∞	∞	∞	∞	-0
5	∞	∞	∞	∞	∞	-0
	0	0	0	0	0	

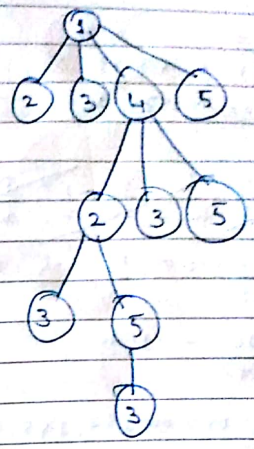
20-02-23

Minimum cost node = Currently exploring node = 2, 5

$\hat{c}(5) = c(R) + A(5,3) + 1 = 28 + 0 + 0 = 28.$

A:

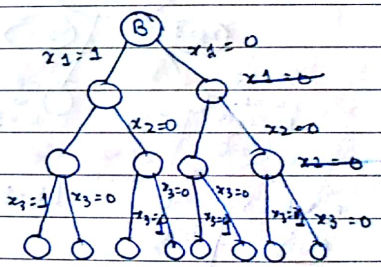
	1	2	3	4	5	
1	∞	∞	∞	∞	∞	-0
2	∞	∞	∞	∞	∞	-0
3	0	∞	∞	∞	∞	-0
4	∞	∞	∞	∞	∞	-0
5	∞	∞	∞	∞	∞	-0
	0	0	0	0	0	



1 → 4 → 2 → 5 → 3

Knapsack Problem - Backtracking

n number of items, each item is associated with weight values and profit value.



Eg: M = 26

$w_i = \{11, 12, 8, 7, 9\}$

$P_i = \{23, 24, 15, 16\}$

$\frac{P_1}{w_1} = \frac{11}{23}$, $\frac{P_2}{w_2} = \frac{12}{24} = \frac{1}{2}$, $\frac{P_3}{w_3} = \frac{8}{4}$

$\frac{P_1}{w_1} = \frac{23}{11} = 2.09$, $\frac{P_2}{w_2} = \frac{24}{12} = 2$, $\frac{P_3}{w_3} = \frac{15}{8} = 1.8$, $\frac{P_4}{w_4} = \frac{16}{9}$

$\frac{P_5}{w_5} = \frac{16}{9} = 1.77$

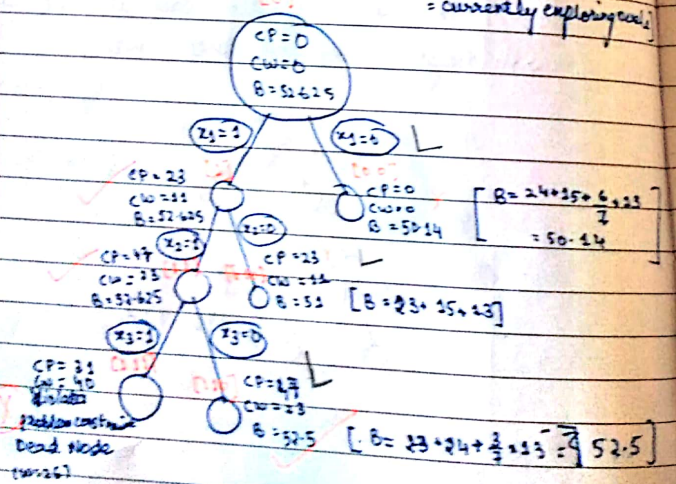
$\frac{P_i}{w_i} = \{2.09, 2.00, 1.87, 1.85, 1.77\}$

Node 1:

Current Profit = 0, Current Weight = 0, (Estimated Profit) Bounding Function = 52.625

By Fractional knapsack, $47 + \frac{3}{8} \times 15 = 52.625$ ($23+24$)

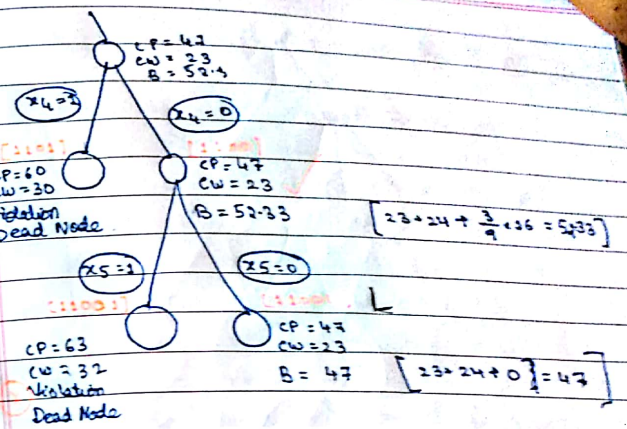
{max profit node (live) = currently exploring node}



$[B = 24 + 15 + \frac{6}{7} \times 23 = 50.14]$

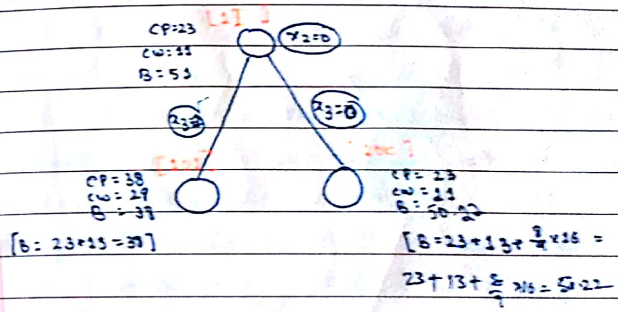
$[B = 23 + 15 + 13]$

$[B = 23 + 24 + \frac{3}{4} \times 13 = 52.5]$



$[23 + 24 + \frac{3}{9} \times 16 = 54.33]$

$[23 + 24 + 0 = 47]$



$[B = 23 + 15 = 39]$

$[B = 23 + 13 + \frac{9}{7} \times 16 = 23 + 13 + \frac{8}{7} \times 16 = 50.22]$

22-04-'18

0/1 Knapsack Problem - Branch and Bound

Branch and Bound Technique Solves only minimization problem, but 0/1 knapsack problem is maximization problem. So, we convert maximization problem into minimization problem.

We define two bounding functions : upper bound and lower bound, they are calculated for each node.

For upper bound, fractional part of items are not allowed; ~~i.e. fractional part of items are not allowed~~

For lower bound, fractional part of items are allowed.

The node which gives least value is currently exploring node.

If for any node, the lower bound value is greater than upper bound value, that node does not generate solution and we are killing it.

To convert into minimization problem, we take upper bound value as negative value.

For each node, we calculate the difference between upper bound and lower bound. Which node gives least value is currently exploring node.

We use two ways for this :

i) LC-BB : Find difference of UB and LB, least value is currently exploring node

ii) FIFO : Exploring the nodes in first in first out manner

$\sum_{i=1}^{n} w_i x_i \leq M$

$\max \sum_{i=1}^{n} p_i x_i$

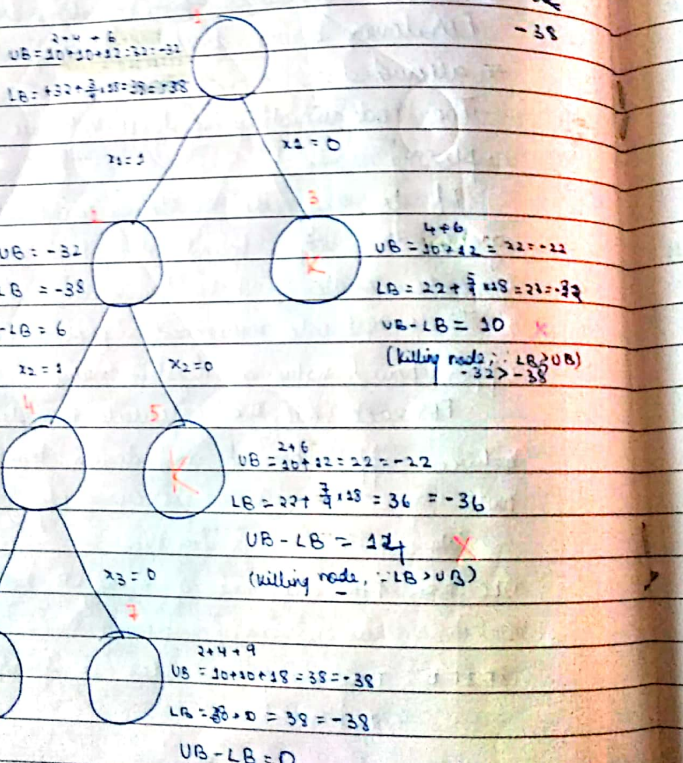
Eg. M = 15, n = 4

$P_i = (10, 10, 12, 18)$

$w_i = (2, 4, 6, 9)$

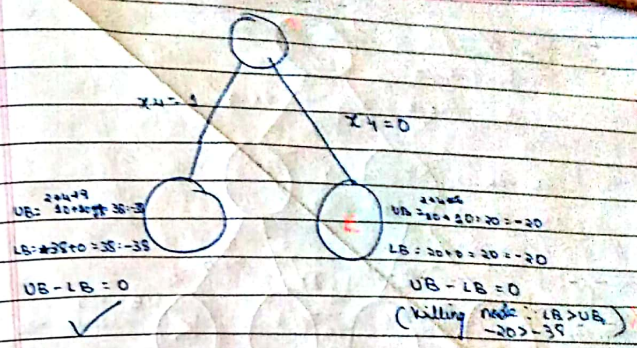
$i = (i_1, i_2, i_3, i_4)$

Least Cost Branch & Bound



Upper = -32

-38



∴ Path: 1 1 0 1 = $x_1 x_2 x_3 x_4$: Soln vector

Eg. M = 15, n = 4

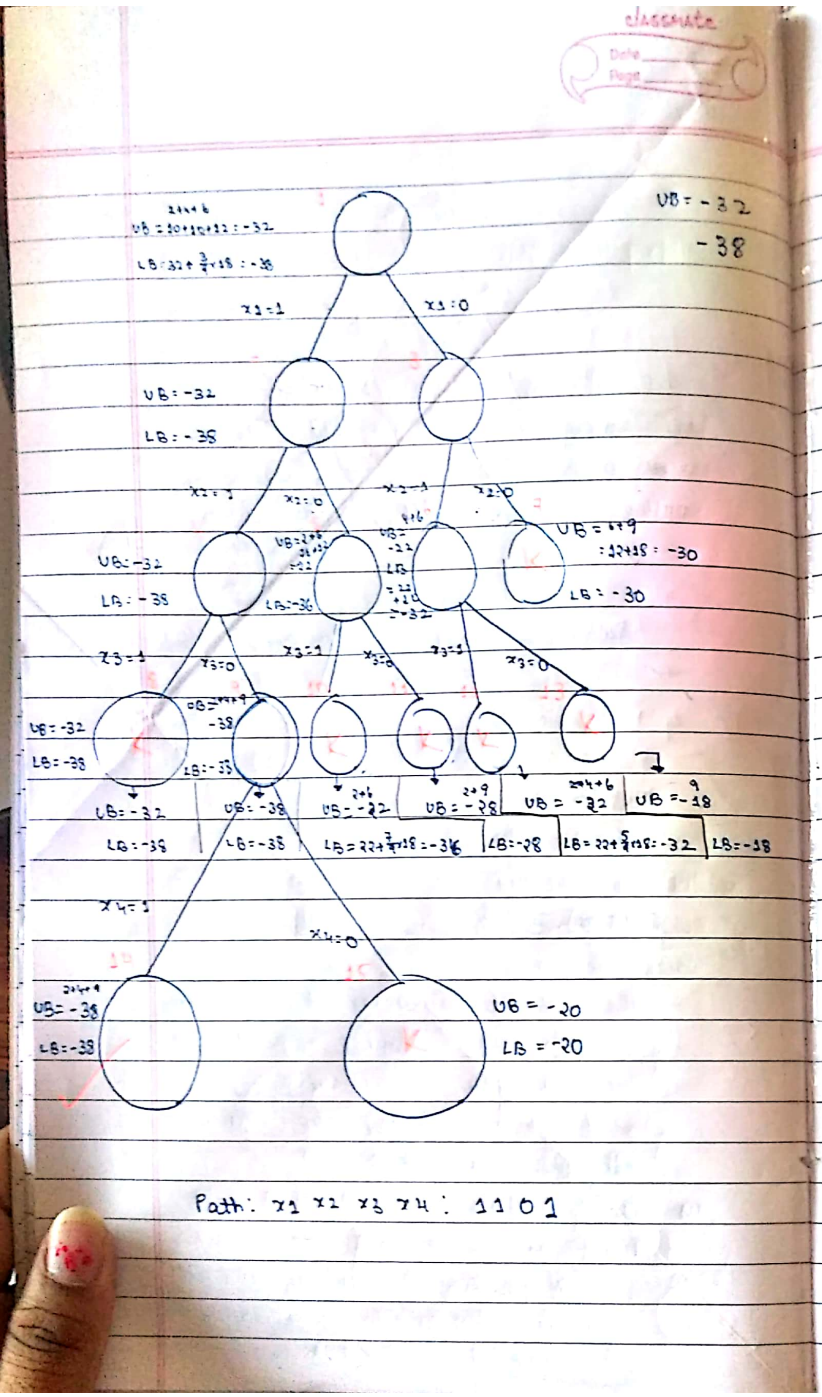
$P_i = (10, 10, 12, 18)$

$w_i = (2, 4, 6, 9)$

$i = (i_1, i_2, i_3, i_4)$

FIFO Branch and Bound

P.T.O →



Path: x1 x2 x3 x4 : 1 1 0 1

UNIT - V

→ N-P Hard and NP Complete Problems

→ Algorithms with time complexity polynomial time

- Linear Search - $O(n)$
- Binary Search - $O(\log n)$
- Insertion Sort - $O(n^2)$
- Merge Sort - $O(n \log n)$
- Matrix Multiplication - $O(n^3)$
- Strassen's Matrix - $O(n^{2.81})$

→ Algorithms with time complexity exponential time

- 0/1 Knapsack Problem - $O(2^n)$
- TVSP - $O(2^n)$
- Graph Colouring - $O(2^n)$
- Hamiltonian Cycle - $O(2^n)$

NP hard and NP complete provides a framework that helps us to solve all problems with exponential time complexity in polynomial time.

If we are not able to write algorithm with polynomial time for algorithms with exponential time, then we must write non-deterministic polynomial algorithm. In non deterministic algorithms we have statements which do not have a clear answer, i.e., yes or no.

If we are not able to write algorithm with polynomial time for algorithms with exponential time then we must find the relation between one problem (that can be solved in polynomial time) and other problems (that cannot be solved in polynomial time)

→ Non deterministic algorithm:

i) For searching algorithm //NP class problem

```
search (a, n, key)
{
    j = choice ();  // 1
    if (a[j] = k)
    {
      write (a[j])
      success ();  // 1
    }
    write (0).
    failure ();  // 1
}
// O(1)
```

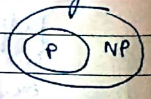
→ Whenever we write non deterministic algorithm we don't know the exact operation of non-deterministic statements, we leave the blanks, they are filled in the future. (Eg. choice(), success(), failure())

We have two classes of problems:

Class P: give polynomial time complexity deterministic algorithm

Class NP: gives polynomial time complexity non deterministic algorithm

Class P problems are subset of class NP, whatever the problems are in class P today they were in class NP yesterday. Whatever problems are in class NP today, they may be in class NP tomorrow. $P \subseteq NP$



→ Satisfiability Problem

CNF

$x_i = (x_1, x_2, x_3)$

$CNF = (x_1 \vee \bar{x_2} \vee x_3) \wedge (\bar{x_1} \vee x_2 \vee \bar{x_3})$ → Propositional Calculus Formula

→ Assume they are two classes. We have to verify all the possibilities for $x_1 x_2 x_3$ which satisfies the statement.

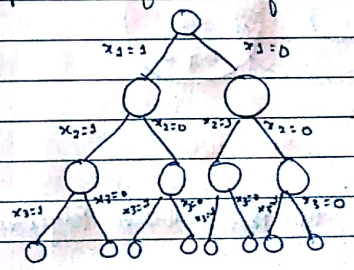
x_1	x_2	x_3
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

$8 = 2^3 = 2^n$ possibilities

Order of satisfiability problem: 2^n

If any person solves satisfiability problem in polynomial time, the all problems with exponential time are solved in polynomial time and related to the satisfiability problem and then solved in exponential time.

State space tree for satisfiability problem.



→ No. of outcomes $= 8 = 2^3 = 2^n$

eg. 0/1 knapsack problem

$n = 3$

$w_1 \quad w_2 \quad w_3$

$x: \{0/1, 0/1, 0/1\}$

Same state space tree for 0/1 knapsack problem and Satisfiability problem. If a researcher solves the satisfiability problem in ext polynomial time, then we relate the satisfiability problem to the 0/1 knapsack problems and then to all other algorithms.

It uses Reduction:

Satisfiability $\propto$ L_1

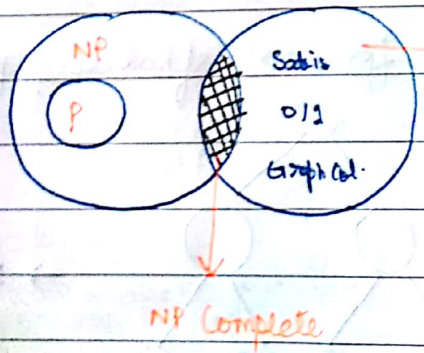
$L_1 \propto L_2$

[$\propto$ → reduction, L_1 → Level 1, L_2 → Level 2]

→ NP Hard and NP complete Problems

If all problems are NP Hard problems. the problems which use non deterministic algorithm are called NP Complete Problems.

NP complete: NP Hard problems which have been solved tomorrow, then include in NP complete



P = NP NP = Non recursive polynomial algorithm