UNIT - IV

Backtracking:

The general method—8 queens problem—Sum of subsets—Graph coloring—Hamiltonian cycle—Knapsack problem.

BACKTRACKING

- It is one of the most general algorithm design techniques.
- Many problems which deal with searching for a set of solutions or for a optimal solution satisfying some constraints can be solved using the backtracking formulation.
- To apply backtracking method, tne desired solution must be expressible as an n-tuple (x1…xn) where xi is chosen from some finite set Si.
- The problem is to find a vector, which maximizes or minimizes a criterion function P(x1….xn).
- The major advantage of this method is, once we know that a partial vector (x1,…xi) will not lead to an optimal solution that (m_{i+1}……..m_n) possible test vectors may be ignored entirely.
- Many problems solved using backtracking require that all the solutions satisfy a complex set of constraints.
- These constraints are classified as:

i) Explicit constraints.
ii) Implicit constraints.

1) Explicit constraints:

Explicit constraints are rules that restrict each Xi to take values only from a given set.

Some examples are,

$Xi \geq 0$ or Si = {all non-negative real nos.}
Xi =0 or 1 or Si={0,1}.
$Li \leq Xi \leq Ui$ or Si= {a: $Li \leq a \leq Ui$}

- All tupules that satisfy the explicit constraint define a possible solution space for I.

2) Implicit constraints:

The implicit constraint determines which of the tuples in the solution space I can actually satisfy the criterion functions.

Algorithm:

```
Algorithm IBacktracking (n)
// This schema describes the backtracking procedure .All solutions are generated in X[1:n]
//and printed as soon as they are determined.
 {
   k=1;
   While (k ≠ 0) do
   {
     if (there remains all untried
     X[k] ∈ T (X[1],[2],…..X[k-1]) and Bk (X[1],…..X[k])) is true ) then
    {
      if(X[1],……X[k] )is the path to the answer node)
     Then write(X[1:k]);
     k=k+1;              //consider the next step.
   }
 else k=k-1;                  //consider backtracking to the previous set.
 }
}
```

- All solutions are generated in X[1:n] and printed as soon as they are determined.
- T(X[1]…..X[k-1]) is all possible values of X[k] gives that X[1],……..X[k-1] have already been chosen.
- B_k(X[1]……X[k]) is a boundary function which determines the elements of X[k] which satisfies the implicit constraint.
- Certain problems which are solved using backtracking method are,

    1. **Sum of subsets.**
    2. **Graph coloring.**
    3. **Hamiltonian cycle.**
    4. **N-Queens problem.**

SUM OF SUBSETS:

- We are given ‘n’ positive numbers called weights and we have to find all combinations of these numbers whose sum is M. this is called sum of subsets problem.

- If we consider backtracking procedure using fixed tuple strategy , the elements X(i) of the solution vector is either 1 or 0 depending on if the weight W(i) is included or not.

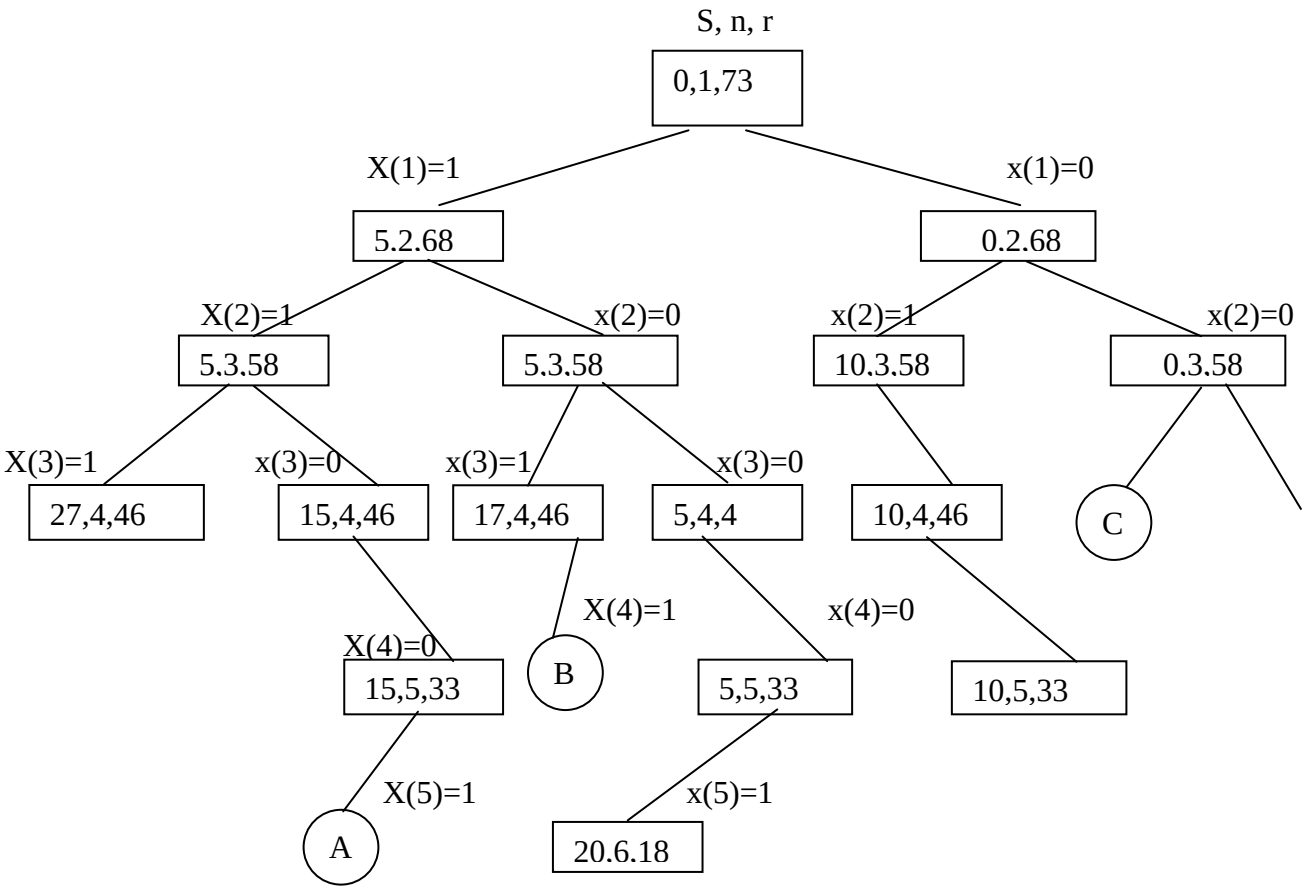
- If the state space tree of the solution, for a node at level I, the left child corresponds to X(i)=1 and right to X(i)=0.

Example:

- Given n=6,M=30 and W(1…6)=(5,10,12,13,15,18).We have to generate all possible combinations of subsets whose sum is equal to the given value M=30.

- In state space tree of the solution the rectangular node lists the values of s, k, r, where s is the sum of subsets,'k' is the iteration and 'r' is the sum of elements after 'k' in the original set.

- The state space tree for the given problem is,



I[st] solution is **A** -> 1 1 0 0 1 0
II[nd] solution is **B** -> 1 0 1 1 0 0

III rd solution is **C** -> 0 0 1 0 0 1

- In the state space tree, edges from level 'i' nodes to 'i+1' nodes are labeled with the values of Xi, which is either 0 or 1.
- The left sub tree of the root defines all subsets containing Wi.
- The right subtree of the root defines all subsets, which does not include Wi.

GENERATION OF STATE SPACE TREE:

- Maintain an array X to represent all elements in the set.
- The value of Xi indicates whether the weight Wi is included or not.
- Sum is initialized to 0 i.e., s=0.
- We have to check starting from the first node.
- Assign X(k)<- 1.
- If S+X(k)=M then we print the subset b'coz the sum is the required output.
- If the above condition is not satisfied then we have to check S+X(k)+W(k+1)<=M. If so, we have to generate the left sub tree. It means W(t) can be included so the sum will be incremented and we have to check for the next k.
- After generating the left sub tree we have to generate the right sub tree, for this we have to check S+W(k+1)<=M.B'coz W(k) is omitted and W(k+1) has to be selected.
- Repeat the process and find all the possible combinations of the subset.

Algorithm:

```
Algorithm sumofsubset(s,k,r)
{
//generate the left child. note s+w(k)<=M since Bk-1 is true.
X{k]=1;
If (S+W[k]=m) then write(X[1:k]); // there is no recursive call here as W[j]>0,1<=j<=n.
Else if (S+W[k]+W[k+1]<=m) then sum of sub (S+W[k], k+1,r- W[k]);
//generate right child and evaluate Bk.
If ((S+ r- W[k]>=m)and(S+ W[k+1]<=m)) then
{
  X{k]=0;
```

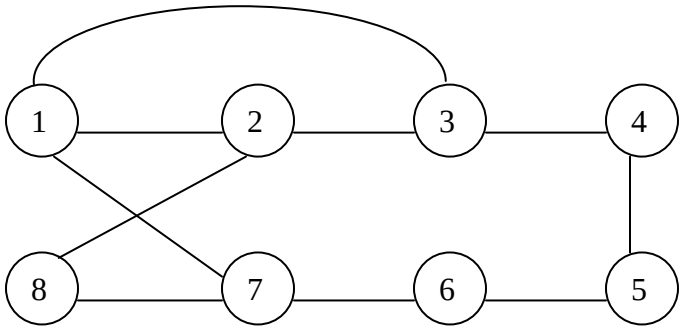
```
  sum of sub (S, k+1, r- W[k]);
}
}
```

HAMILTONIAN CYCLES:

- Let G=(V,E) be a connected graph with 'n' vertices. A HAMILTONIAN CYCLE is a round trip path along 'n' edges of G which every vertex once and returns to its starting position.

- If the Hamiltonian cycle begins at some vertex V1 belongs to G and the vertex are visited in the order of V1,V2…….Vn+1,then the edges are in E,1<=I<=n and the Vi are distinct except V1 and Vn+1 which are equal.

- Consider an example graph G1.



The graph G1 has Hamiltonian cycles:

->1,3,4,5,6,7,8,2,1 and
->1,2,8,7,6,5,4,3,1.

- The backtracking algorithm helps to find Hamiltonian cycle for any type of graph.

Procedure:

1. Define a solution vector X(Xi……..Xn) where Xi represents the I th visited vertex of the proposed cycle.

2. Create a cost adjacency matrix for the given graph.

3. The solution array initialized to all zeros except X(1)=1,b'coz the cycle should start at vertex '1'.

4. Now we have to find the second vertex to be visited in the cycle.

5. The vertex from 1 to n are included in the cycle one by one by checking 2 conditions,

1.There should be a path from previous visited vertex to current vertex.

2.The current vertex must be distinct and should not have been visited earlier.

6. When these two conditions are satisfied the current vertex is included in the cycle, else the next vertex is tried.

7. When the nth vertex is visited we have to check, is there any path from nth vertex to first 8vertex. if no path, the go back one step and after the previous visited node.

8. Repeat the above steps to generate possible Hamiltonian cycle.

Algorithm:(Finding all Hamiltonian cycle)

```
Algorithm Hamiltonian (k)
{
 Loop
        Next value (k)
If (x (k)=0) then return;
{
  If k=n then
     Print (x)
Else
Hamiltonian (k+1);
End if

}
Repeat
}

Algorithm Nextvalue (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 distinction.
  If (j=k) then          //if true then the vertex is distinct.
    If ((k<n) or ((k=n) and G [X [n], X [1]] ≠ 0)) then return;
}
} Until (false);
}
```

8-QUEENS PROBLEM:

This 8 queens problem is to place n-queens in an 'N*N' matrix in such a way that no two queens attack each otherwise no two queens should be in the same row, column, diagonal.

Solution:

- The solution vector X (X1…Xn) represents a solution in which Xi is the column of the th row where I th queen is placed.
- First, we have to check no two queens are in same row.
- Second, we have to check no two queens are in same column.
- The function, which is used to check these two conditions, is [I, X (j)], which gives position of the I th queen, where I represents the row and X (j) represents the column position.
- Third, we have to check no two queens are in it diagonal.
- Consider two dimensional array A[1:n,1:n] in which we observe that every element on the same diagonal that runs from upper left to lower right has the same value.
- Also, every element on the same diagonal that runs from lower right to upper left has the same value.
- Suppose two queens are in same position (i,j) and (k,l) then two queens lie on the same diagonal , if and only if |j-l|=|I-k|.

STEPS TO GENERATE THE SOLUTION:

- Initialize x array to zero and start by placing the first queen in k=1 in the first row.
- To find the column position start from value 1 to n, where 'n' is the no. Of columns or no. Of queens.
- If k=1 then x (k)=1.so (k,x(k)) will give the position of the k th queen. Here we have to check whether there is any queen in the same column or diagonal.
- For this considers the previous position, which had already, been found out. Check whether
  X (I)=X(k) for column |X(i)-X(k)|=(I-k) for the same diagonal.
- If any one of the conditions is true then return false indicating that k th queen can't be placed in position X (k).
- For not possible condition increment X (k) value by one and precede d until the position is found.

- If the position X (k)≤ n and k=n then the solution is generated completely.
- If k<n, then increment the 'k' value and find position of the next queen.
- If the position X (k)>n then k th queen cannot be placed as the size of the matrix is 'N*N'.
- So decrement the 'k' value by one i.e. we have to back track and after the position of the previous queen.

Algorithm:

```
Algorithm place (k,I)
//return true if a queen can be placed in k th row and I th column. otherwise it returns //
//false .X[] is a global array whose first k-1 values have been set. Abs® returns the
//absolute value of r.
{
  For j=1 to k-1 do
    If ((X [j]=I)            //two in same column.
    Or (abs (X [j]-I)=Abs (j-k)))
Then return false;
Return true;
}
```

Algorithm Nqueen (k,n)

```
//using backtracking it prints all possible positions of n queens in 'n*n' chessboard. So
 //that they are non-tracking.
{
    For I=1 to n do
       {
         If place (k,I) then
           {
             X [k]=I;
              If (k=n) then write (X [1:n]);
               Else nquenns(k+1,n)    ;
         }
     }
}
```

Example: 4 queens.
Two possible solutions are

	Q		
			Q
Q			
		Q	

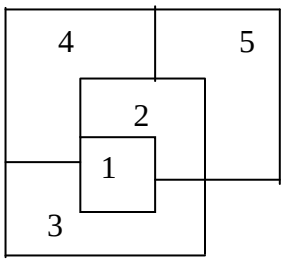
		Q	
Q			
			Q
	Q		

Solutin-1	Solution 2
(2 4 1 3)	(3 1 4 2)

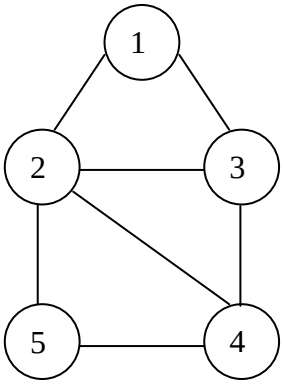
GRAPH COLORING:

- Let ‘G’ be a graph and ‘m’ be a given positive integer. If the nodes of ‘G’ can be colored in such a way that no two adjacent nodes have the same color. Yet only ‘M’ colors are used. So it’s called M-color ability decision problem.
- The graph G can be colored using the smallest integer ‘m’. This integer is referred to as chromatic number of the graph.
- A graph is said to be planar iff it can be drawn on plane in such a way that no two edges cross each other.
- Suppose we are given a map then, we have to convert it into planar. Consider each and every region as a node. If two regions are adjacent then the corresponding nodes are joined by an edge.

Consider a map with five regions and its graph.



1 is adjacent to 2, 3, 4.
2 is adjacent to 1, 3, 4, 5
3 is adjacent to 1, 2, 4
4 is adjacent to 1, 2, 3, 5
5 is adjacent to 2, 4



Steps to color the Graph:

- First create the adjacency matrix graph(1:m,1:n) for a graph, if there is an edge between i,j then C(i,j) = 1 otherwise C(i,j) =0.

- The Colors will be represented by the integers 1,2,…..m and the solutions will be stored in the array X(1),X(2),……..,X(n) ,X(index) is the color, index is the node.

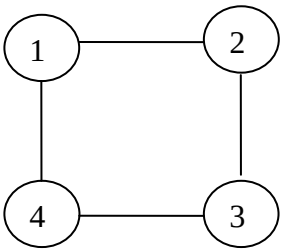
- He formula is used to set the color is,
  X(k) = (X(k)+1) % (m+1)

- First one chromatic number is assigned ,after assigning a number for 'k' node, we have to check whether the adjacent nodes has got the same values if so then we have to assign the next value.

- Repeat the procedure until all possible combinations of colors are found.

- The function which is used to check the adjacent nodes and same color is,
  If((Graph (k,j) == 1) and X(k) = X(j))

Example:



N= 4
M= 3

Adjacency Matrix:

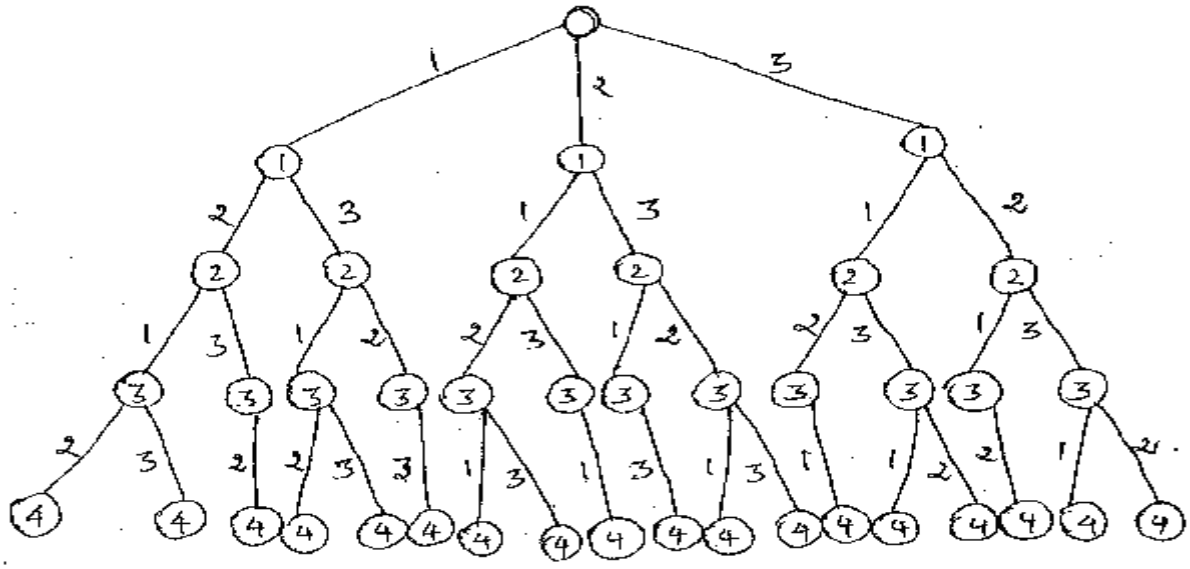
$$\begin{vmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{vmatrix}$$

→ Problem is to color the given graph of 4 nodes using 3 colors.

→Node-1 can take the given graph of 4 nodes using 3 colors.

→ The state space tree will give all possible colors in that ,the numbers which are inside the circles are nodes ,and the branch with a number is the colors of the nodes.

State Space Tree:



Algorithm:

Algorithm mColoring(k)

// the graph is represented by its Boolean adjacency matrix G[1:n,1:n] .All assignments //of 1,2,…….,m to the vertices of the graph such that adjacent vertices are assigned //distinct integers are printed. 'k' is the index of the next vertex to color.

```
{
repeat
{
   // generate all legal assignment for X[k].
  Nextvalue(k);    // Assign to X[k] a legal color.
       If (X[k]=0) then return;          // No new color possible.
      If (k=n) then                 // Almost 'm' colors have been used to color the 'n' vertices
              Write(x[1:n]);
     Else mcoloring(k+1);
}until(false);
}
```

Algorithm Nextvalue(k)

// X[1],……X[k-1] have been assigned integer values in the range[1,m] such that //adjacent values have distinct integers. A value for X[k] is determined in the //range[0,m].X[k] is assigned the next highest numbers color while maintaining //distinctness form the adjacent vertices of vertex K. If no such color exists, then X[k] is 0.

```
{
```

```
  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
          {
             // Check if this color is distinct from adjacent color.
          If((G[k,j] ≠ 0)and(X[k] = X[j]))
             // If (k,j) is an edge and if adjacent vertices have the same color.
          Then break;
          }

       if(j=n+1) then return;     //new color found.
   } until(false);    //otherwise try to find another color.
}
```

→ The time spent by Nextvalue to determine the children is $\theta(mn)$

→Total time is = $\theta(m^n n)$.

Knapsack Problem using Backtracking:

- The problem is similar to the zero-one (0/1) knapsack optimization problem is dynamic programming algorithm.

- We are given 'n' positive weights Wi and 'n' positive profits Pi, and a positive number 'm' that is the knapsack capacity, the is problem calls for choosing a subset of the weights such that,

$$\sum_{1 \le i \le n} WiXi \le m \text{ and } \sum_{1 \le i \le n} PiXi \text{ is Maximized.}$$

Xi →Constitute Zero-one valued Vector.

- The Solution space is the same as that for the sum of subset's problem.

- Bounding functions are needed to help kill some live nodes without expanding them. A good bounding function for this problem is obtained by using an upper bound on the value of the best feasible solution obtainable by expanding the given live node.

- The profits and weights are assigned in descending order depend upon the ratio.

(i.e.) Pi/Wi ≥P(I+1) / W(I+1)

Solution :

- After assigning the profit and weights ,we have to take the first object weights and check if the first weight is less than or equal to the capacity, if so then we include that object (i.e.) the unit is 1.(i.e.) K→ 1.

- Then We are going to the next object, if the object weight is exceeded that object does not fit. So unit of that object is '0'.(i.e.) K=0.
- Then We are going to the bounding function ,this function determines an upper bound on the best solution obtainable at level K+1.

- Repeat the process until we reach the optimal solution.

Algorithm:

Algorithm Bknap(k,cp,cw)

// 'm' is the size of the knapsack; 'n' → no.of weights & profits. W[]&P[] are the //weights & weights. P[I]/W[I] ≥ P[I+1]/W[I+1].
//fw→Final weights of knapsack.
//fp→ final max.profit.
//x[k] = 0 if W[k] is not the knapsack,else X[k]=1.

```
{
     // Generate left child.
      If((W+W[k] ≤m) then
      {
           Y[k] =1;
            If(k<n) then Bnap(k+1,cp+P[k],Cw +W[k])
              If((Cp + p[w] > fp) and (k=n)) then

               {
                 fp = cp + P[k];
                 fw = Cw+W[k];
                for j=1 to k do X[j] = Y[j];
               }
  }

 if(Bound(cp,cw,k) ≥fp) then
 {
     y[k] = 0;
   if(k<n) then Bnap (K+1,cp,cw);
  if((cp>fp) and (k=n)) then
    {
        fp = cp;
         fw = cw;
           for j=1 to k do X[j] = Y[j];
```

```
      }
   }
}
```

Algorithm for Bounding function:

```
Algorithm Bound(cp,cw,k)
// cp→ current profit total.
//cw→ current weight total.
//k→the index of the last removed item.
//m→the knapsack size.

{
    b=cp;
    c=cw;
    for I =- k+1 to n do
  {
        c= c+w[I];
       if (c<m) then b=b+p[I];
          else return b+ (1-(c-m)/W[I]) * P[I];
}
return b;
}
```

Example:

M= 6 Wi = 2,3,4 4 2 2

N= 3 Pi = 1,2,5 Pi/Wi (i.e.) 5 2 1

Xi = 1 0 1
The maximum weight is 6

The Maximum profit is (1*5) + (0*2) + (1*1)
→ 5+1
→ 6.

Fp = (-1)

- $1 \leq 3$ & $0+4 \leq 6$
  cw = 4,cp = 5,y(1) =1
  k = k+2

- $2 \leq 3$ but 7>6
  so y(2) = 0

- So bound(5,4,2,6)

B=5
C=4
I=3 to 3
C=6
$6 \neq 6$
So return 5+(1-(6-6))/(2*1)

- 5.5 is not less than fp.
  So, k=k+1 (i.e.) 3.

  3=3 & $4+2 \leq 6$

  cw= 6,cp = 6, y(3)=1.
  K=4.

- If 4> 3 then

  Fp =6,fw=6,k=3 ,x(1) 1 0 1
  The solution Xi → 1 0 1

  Profit → 6
  Weight →6.