DESIGN & ANALYSIS OF ALGORITHMS

UNIT – V

5.1. Introduction: Backtracking

5.1.1. General method

5.2. Applications

5.2.1. n-Queen Problem

5.2.2. Sum of Subsets Problem

5.2.3. Graph Coloring

5.2.4. Hamiltonian Cycles

5.3. Introduction: Branch and Bound

5.3.1. General method

5.4. Applications

5.4.1. Travelling Sales Person Problem

5.4.2. 0/1 Knapsack Problem-LCBB

5.4.3. 0/1 Knapsack Problem-FIFOBB.

Introduction:

In case of Greedy and Dynamic Programming techniques, we will use Brute Force approach. It means we will evaluate all possible solution, among which, we can select one solution as optimal solution.

In back tracking technique, we will get same optimal solution with a less number of steps. So, by using back tracking technique, we will solve problems in an efficient way, when compared to other methods like Greedy and Dynamic Programming.

The name back track was first coined by D.H.Lehamen in the 1950's. early workers who studied the process were R.J.Walker, who gave an algorithmic account of it in 1960 and S.Golomb and L.Baumart who presented a very general description of it as well as a variety of applications.

In this Back Tracking Method

1. **The Desired solution is expressible as an n-tuple ($x_1,x_2,x_3,\ldots,x_n$), where x_i are chosen from same finite set S_i.**
2. **The solution maximizes or minimizes or satisfies a criterion function $F(x_1,x_2,x_3,\ldots,x_n)$.**

The major advantage of back tracking method is, if a partial solution ($x_1,x_2,x_3,\ldots,x_i$) can't to lead optimal solutions then (x_{i+1}, ….x_n) solution may be ignored completely.

The problem can be categorized in to three types

For Instance-for a problem P

Let C be the set constraints for P

Let D be the set containing all solutions satisfying C then finding whether there is any feasible solution?. – is the decision problem.

What is the best solution?. – is the optimization problem

Listing all the feasible solutions- is the enumeration problem

The basic idea of the back tracking is to build up a vector. One component at a time and to test whether the vector being formed has any chance of success.

The major advantage of this algorithm is that we can realize the fact that the partial vector generates does not lead to an optimal solution. In such a situation, the vector can be ignored.

Back tracking algorithm determines the solution by systematically searching the solution space(i,e set of all feasible solutions) for the given problem.

Back tracking is a DFS with some bounding function. All solutions using back tracking are required to satisfy a complex set of constraints. The constraints may be implicit or explicit.

Explicit constraints: these are rules, which restrict each x_i to take an values only from the given set.

Example: in Knapsack problem the explicit constraints are

1. $x_i = 0$ or 1

2. $0 \leq x_i \leq 1$

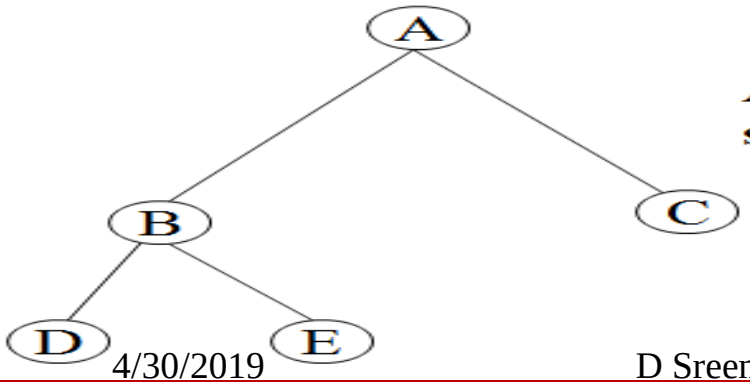
Implicit constraints: these are rules, which determine which of the tuples in the solution space , satisfy the criterion function.

Example: in 4-Queens problem the implicit constraints are no two Queens can be on the same column, same row and same diagonal.

Basic Terminology:

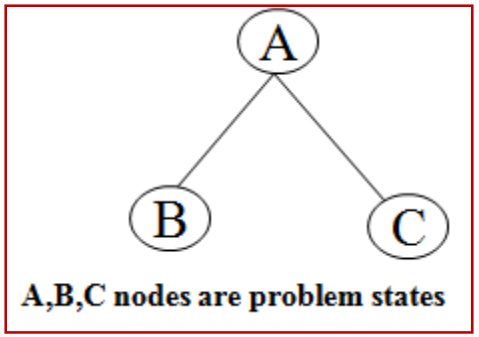
1. Criterion Function: it is a function $P(x_1,x_2,x_3,\ldots,x_n)$ which needs to be maximized or minimized for a given function.

2. Solution Space: all tuples that satisfy the explicit constraints define a possible solution space for a particular instance 'I' of the problem

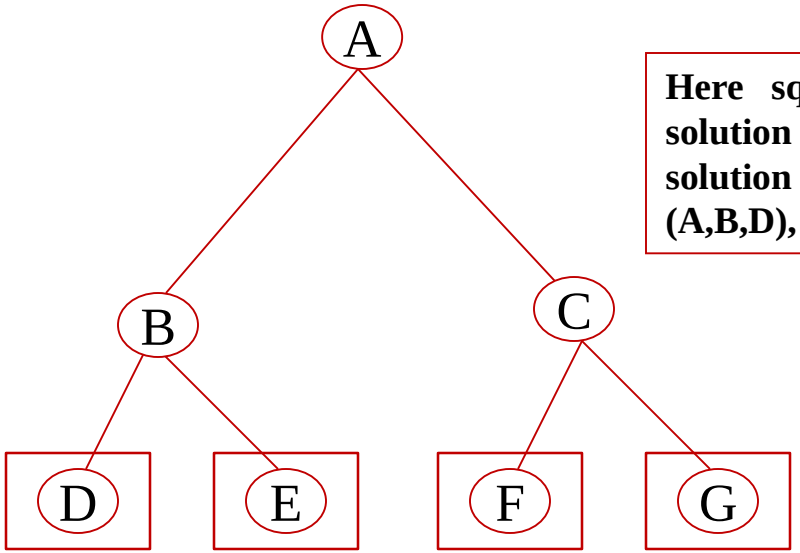


ABD, ABE, AC are the tuples in solution Space

3. Problem State: **Each node in the tree organization defines a problem state.**

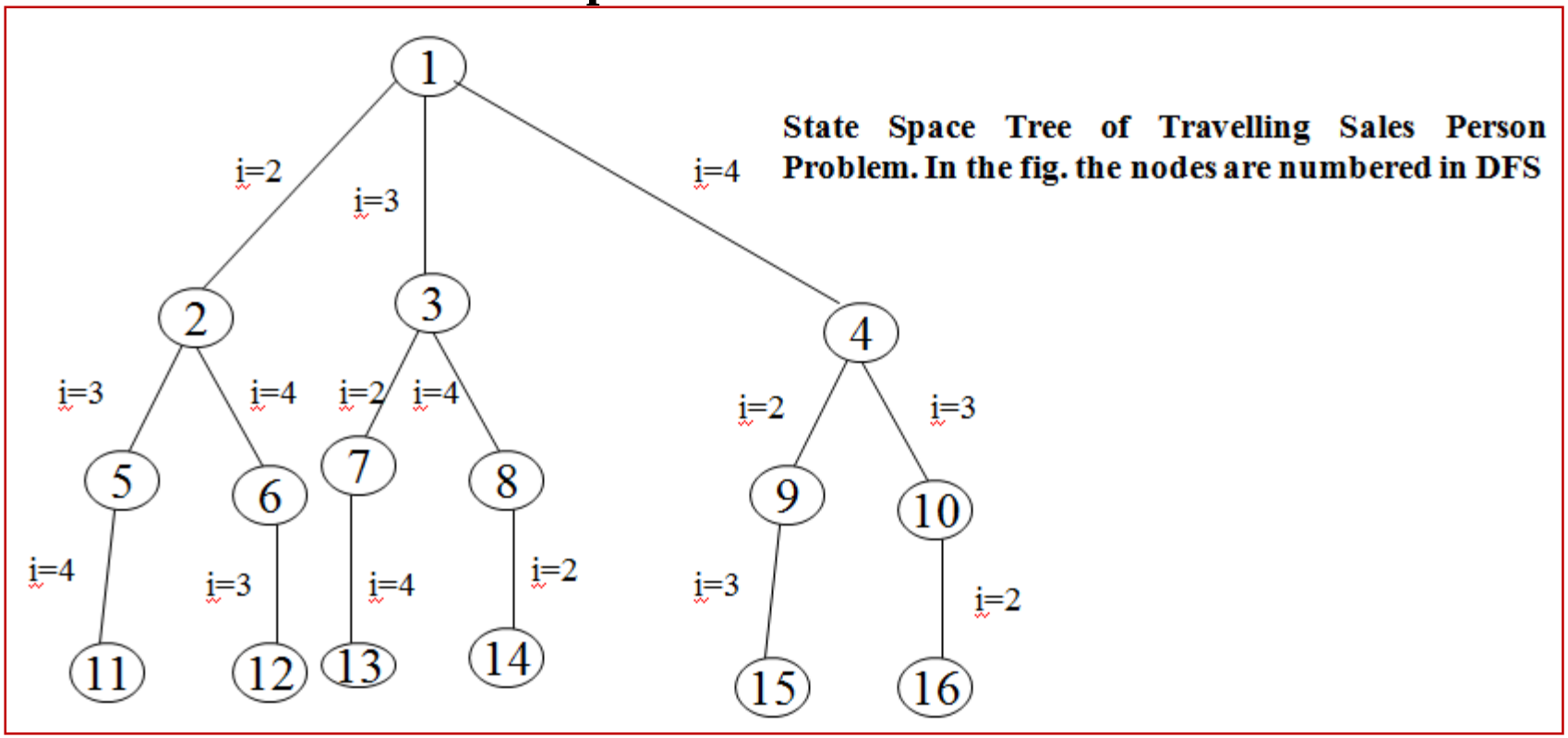


4. Solution States: **these are those problem states S for which the path from the root to S define a tuple in the solution space.**



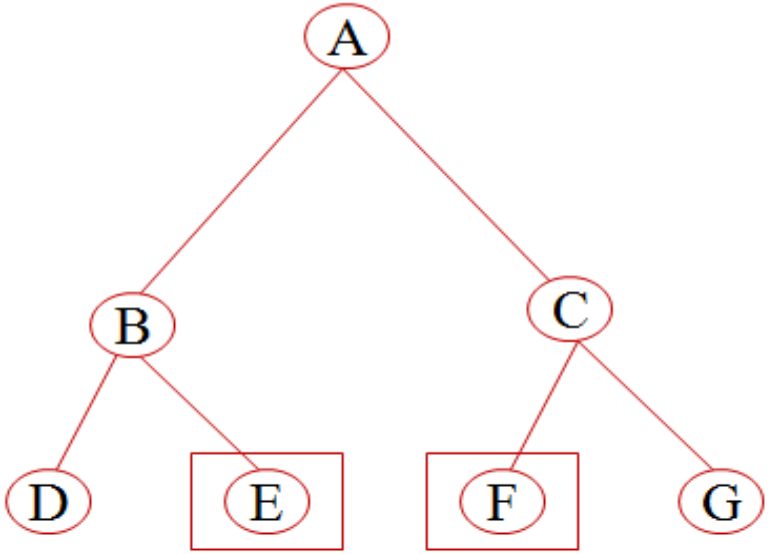
Here square nodes indicates solution. For the solution space, there exist 4 solution states. These solution states represented in the form of tuples (A,B,D), (A,B,E), (A,C,F) and (A,C,G)

5. State Space Tree: if we represent solution space in the form of a tree then the tree is referred as the state space tree.



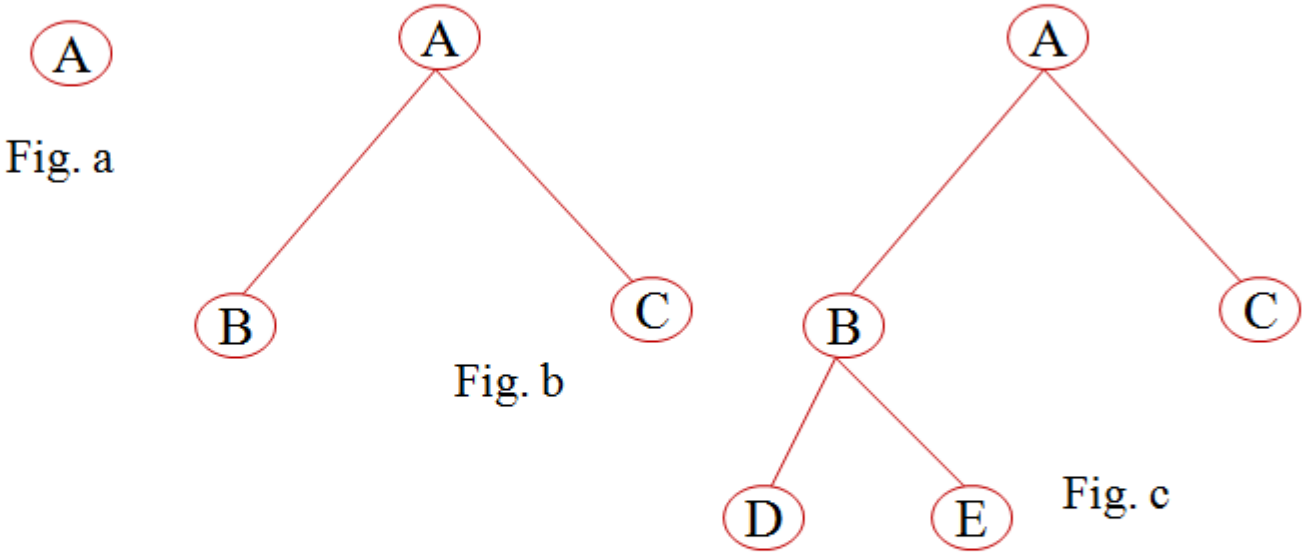
State Space Tree of Travelling Sales Person Problem. In the fig. the nodes are numbered in DFS

6. Answer States: **these solution states S for which the path from the root to S define a tuple which is a member of the set of solutions of the problem**



Here E **and** F **are answer states. (A,B,E) and (A,C,F) are solution states**

7. Live States: a node which has been generated and all of whose children have not yet been generated

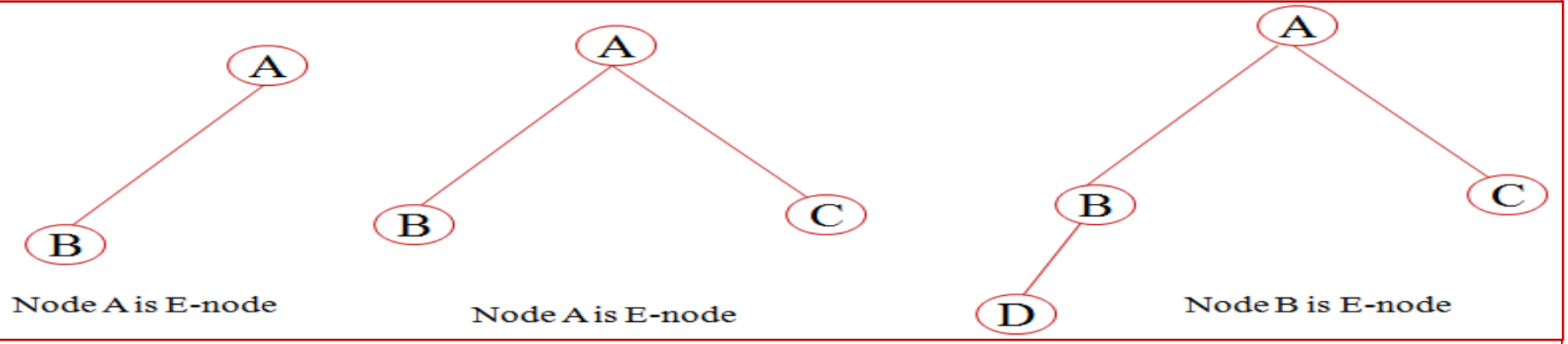


In fig.(a) node A is called live node since the children of node A have not yet been generated.

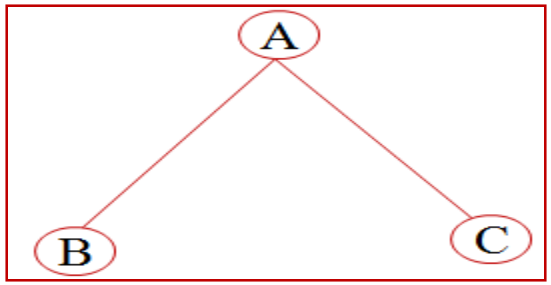
In fig.(b) node A is not a live node but node B an C are live nodes

In fig.(c) nodes D,E,C are live nodes because the children of these nodes are not yet been generated.

8. E-Node: **the live nodes whose children are currently being generated is called the E-node(node being expanded)**



9. Dead node: **it is generated node. That is either not to be expanded further or one for which all of its children have been generated.**



Nodes A,B,C are dead nodes. Since node A's children generated and node B,C are not expanded.

General Method or Control Abstraction:

Let $(x_1,x_2,x_3,\ldots,x_k)$ be the path from the root to a node in a state space tree. Let $T(x_1,x_2,x_3,\ldots,x_{k+1})$ be the set of all possible values for x_{k+1} such that $(x_1,x_2,x_3,\ldots,x_{k+1})$ is also to a problem state.

We shall assume the existence of bounding functionsB_{i+1}(expressed as predicates) such that $B_{i+1}(x_1,x_2,x_3,\ldots,x_{k+1})$ is false for the path $(x_1,x_2,x_3,\ldots,x_{k+1})$ from the root node to a problem state only if the path can not be extended to reach an answer node.

Thus the candidates for position i+1 of the solution vector x(1:n) are those values which are generated by T and satisfy B_{k+1}.

Algorithm Back_Track(k)

```
// this schema describes the back tracking process using recursion. On entering, the first k-1 values
//x[1], x[2]……..,x[k-1] of the solution vector x[1:n] have been assigned. X[] and n are global
{
   For(each x[k] ∈ T(x[1], x[2],….,x[k-1]) do
   {
       If (B_k(x[1], x[2],….,x[k-1]) # 0) then
       {
          If (x[1], x[2],….,x[k-1]) is a path to an answer node) then
             Write(x[i:k]);
          If(k<n) then
             Back_Track(k+1);
       }
   }
}
```

The Back Tracking Applications

1. N-Queen Problem
2. Sum of Subsets Problem
3. Graph Coloring
4. Hamiltonian Cycles

5.2.1. N-Queen Problem (4-Queen's and 8-Queen's Problems)

Consider an nxn chess board. Let there are n Queens. These n Queens are to be placed on nxn chess board so that no two Queens are on the same column, same row or same diagonal.

1. 4-Queen's Problem:

Consider a 4x4 chess board. Let there are 4 Queen's. the objective is to place the 4-Queen's on 4x 4 chess board in such a way that no two queens should be placed in the same row, same column and same diagonal position

The explicit constraint is 4 Queen's are to be placed on 4x4 chess board in 4^4 ways.

The implicit constraint is no two queens should be placed in the same row, same column and same diagonal position.

Let $\{x_1,x_2,x_3,x_4\}$ be the solution vector where x_i, column number on which the queen i is placed.

First queen Q_1 is placed in the first row and first column.

Q1			

The second queen should not be placed in first row and first column.

It should be placed in second row and in second, third, or fourth column.

If you placed in 2ndcolumn, both will be the same diagonal, so place it in 3rd column.

Q_1	X	X	X
X	X	Q_2	
X			
X			

Q_1	X	X	X
X	X	Q_2	X
X	X	X	X
X		X	

Q_1	X	X	X
X	X	Q_2	X
X	X	X	X
X	Q_3	X	X

or

Q_1	X	X	X
X	X	Q_2	X
X	X	X	X
X	X	X	Q_3

So, go back to Q2 and place it some where else

Q_1	X	X	X
X	X	X	Q_2
X		X	X
X	X		X

Now 2nd queen may be placed in 4th column, then Q3 is may be placed in 2nd column or 3rd column

If Q3 placed in 2ndcolumn and the remaining position for Q4 is 3rd column, there will be diagonal attack from Q3. so, go back, remove Q3 and place it in the next column. But it is also not possible. So, move back to Q2, place it in next column., but it is not possible. So, go back to Q1 and move it to the next column. It can be shown as

X	Q_1	X	X
X	X	X	
	X		X
	X		

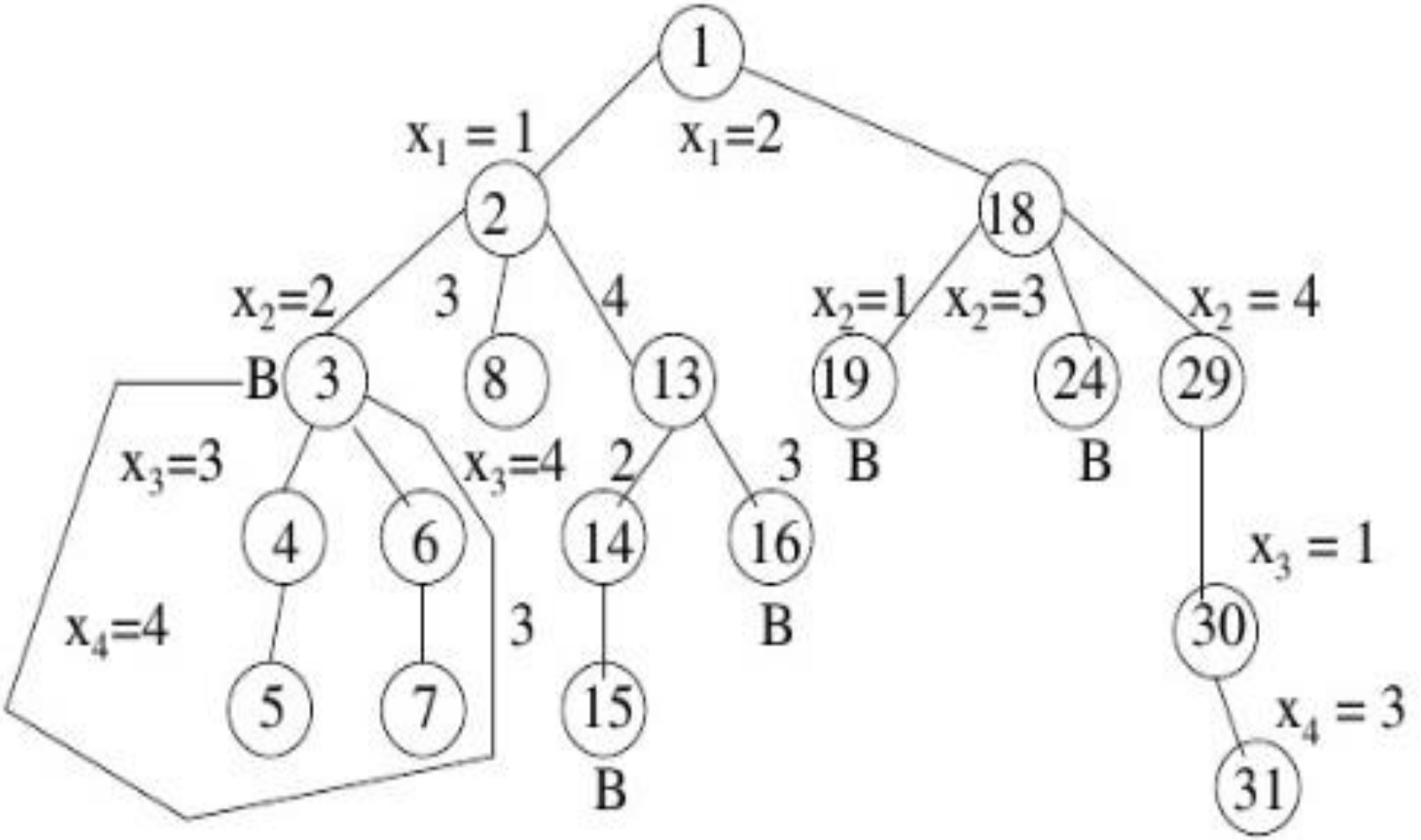
X	Q_1	X	X
X	X	X	Q_2
	X	X	X
	X		X

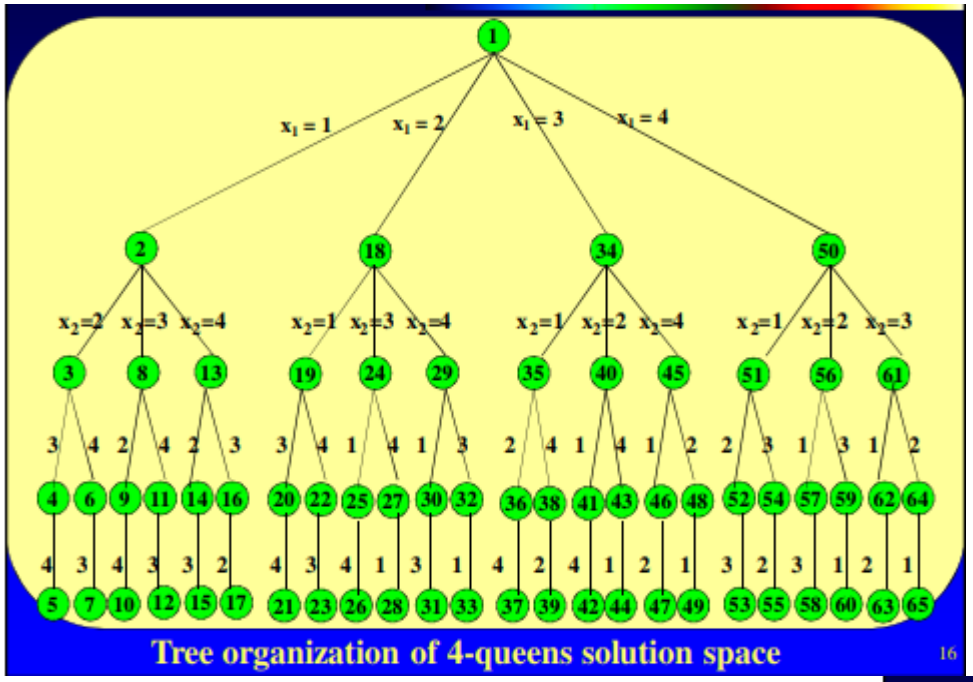
X	Q_1	X	X
X	X	X	Q_2
Q_3	X	X	X
X	X		X

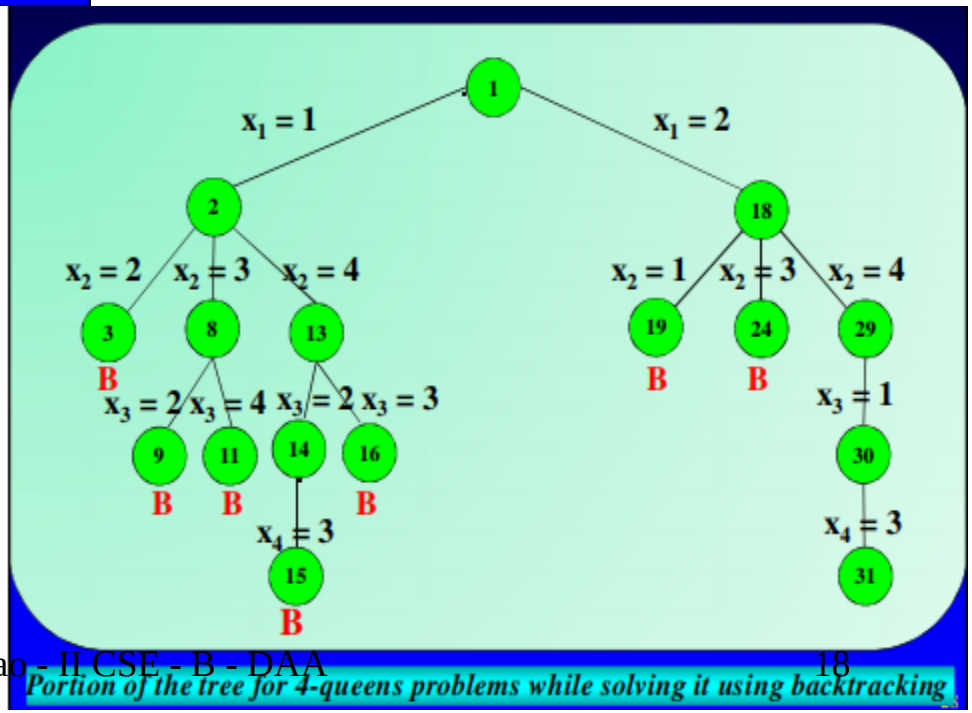
X	Q_1	X	X
X	X	X	Q_2
Q_3	X	X	X
X	X	Q_4	X

Hence the solution to 4-Queen's problem is x_1=2, x_2=4, x_3=1 and x_4=3 (2, 4, 1, 3)

State space tree: 4 Queens problem




1
x1 = 1
x1 = 2
x1 = 3
x1 = 4
2
18
34
50
x2=2 x2=3 x2=4
x2=1 x2=3 x2=4
x2=1 x2=2 x2=4
x2=1 x2=2 x2=3
3 8 13 19 24 29 35 40 45 51 56 61
3 4 2 4 2 3 3 4 1 4 1 3 2 4 1 4 1 2 2 3 1 3 1 2
4 6 9 11 14 16 20 22 25 27 30 32 36 38 41 43 46 48 52 54 57 59 62 64
4 3 4 3 3 2 4 3 4 1 3 1 4 2 4 1 2 1 3 2 3 1 2 1
5 7 10 12 15 17 21 23 26 28 31 33 37 39 42 44 47 49 53 55 58 60 63 65
Tree organization of 4-queens solution space
16


1
x1 = 1
x1 = 2
2
18
x2 = 2 x2 = 3 x2 = 4
x2 = 1 x2 = 3 x2 = 4
3 8 13 19 24 29
B B B
x3 = 2 x3 = 4 x3 = 2 x3 = 3
x3 = 1
9 11 14 16 30
B B B
x4 = 3
x4 = 3
15 31
B
Portion of the tree for 4-queens problems while solving it using backtracking

2. 8-Queen's Problem:

Consider a 8x8 chess board. Let there are 8 Queen's. the objective is to place the 8-Queen's on 8x8 chess board in such a way that no two queens should be placed in the same row, same column and same diagonal position

			Q_1				
					Q_2		
							Q_3
	Q_4						
						Q_5	
Q_6							
		Q_7					
				Q_8			

The solution is $(x_1,x_2,x_3,x_4,x_5,x_6,x_7,x_8)=(4,6,8,2,7,1,3,5)$

```
Algorithm NQueens(k,n)
{
  for i:= 1 to n do
  {
    if(place(k,i)) then
   {
     x[k]:=i;
     If(k=n) // column number = order of matrix
        write(x[1:n]);
     else
        NQueens(k+1,n);
   }
 }
}
```

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

Time Complexity:

The solution space tree of 8-Queen's problem contains 88 tuples.

After imposing implicit constraint, the size of the solution space is reduced to 8! Tuples.

Hence the time complexity is O(8!)

For n-Queen's problem, it is O(n!).

5.2.2. Sum of Subsets Problem

Suppose we are given n distinct positive numbers (usually called weights) and we desire to find all combinations of these numbers whose sum are m. this is called the Sum of Subsets Problem.

Problem Statement:

Let S = $\{s_1, s_2, \ldots, s_n\}$ be a set of n positive integers, then we have to find a subset whose sum is equal to given positive integer d.

it is always convenient to sort the sets elements in an ascending order. That is, $s_1 \leq s_2 \leq \ldots\ldots \leq s_n$.

We consider a back tracking solution using the fixed size tuple strategy. In this case the element x_i of the solution vector is either 1 or 0 depending on whether the weight w_i is included or not.

For a node at level i the left child corresponds to x_i=1 and the right child to x_i=0.

A simple choice for the bounding function is $B_k(x_1 \ldots x_k)$

$$\sum_{i=1}^{k} w_i x_i + \sum_{i=k+1}^{n} w_i \geq M$$

Clearly $x_1 \ldots x_k$ can not lead to answer node if this condition is not satisfied.

The bounding functions can be strengthened if we assume the w_i 's are initially in non decreasing order.

In this case $x_1 \ldots x_k$ can not lead to an answer node if $\sum_{i=1}^{k} w_i x_i + w_{k+1} > M$

The bounding functions we use are

Therefore $B_k(x_1 \ldots x_k)$ = true iff $\sum_{i=1}^{k} w_i x_i + \sum_{i=k+1}^{n} w_i \geq M$

and $\sum_{i=1}^{k} w_i x_i + w_{k+1} \leq m$

Algorithm or Procedure:

Let S be a set of elements and d is the expected sum of subsets then

Step 1: 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 we have reached the end of the set then back track through the subset untill we find the most suitable value.

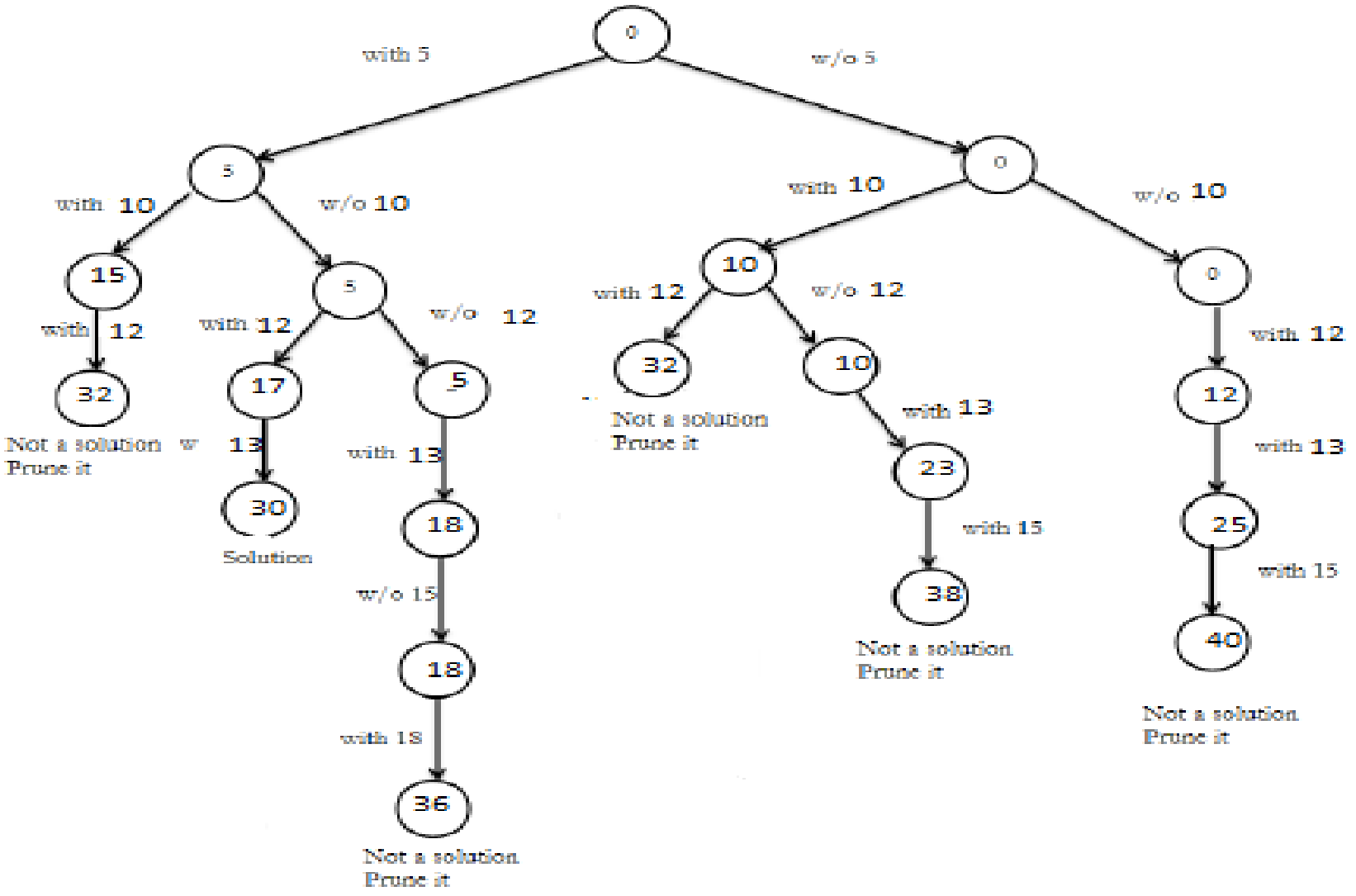
Step 5: if the sub set is feasible then repeat step 2.

Step 6: if we have visited all the elements without finding a suitable subset and if no backtracking is possible then stop without solution.

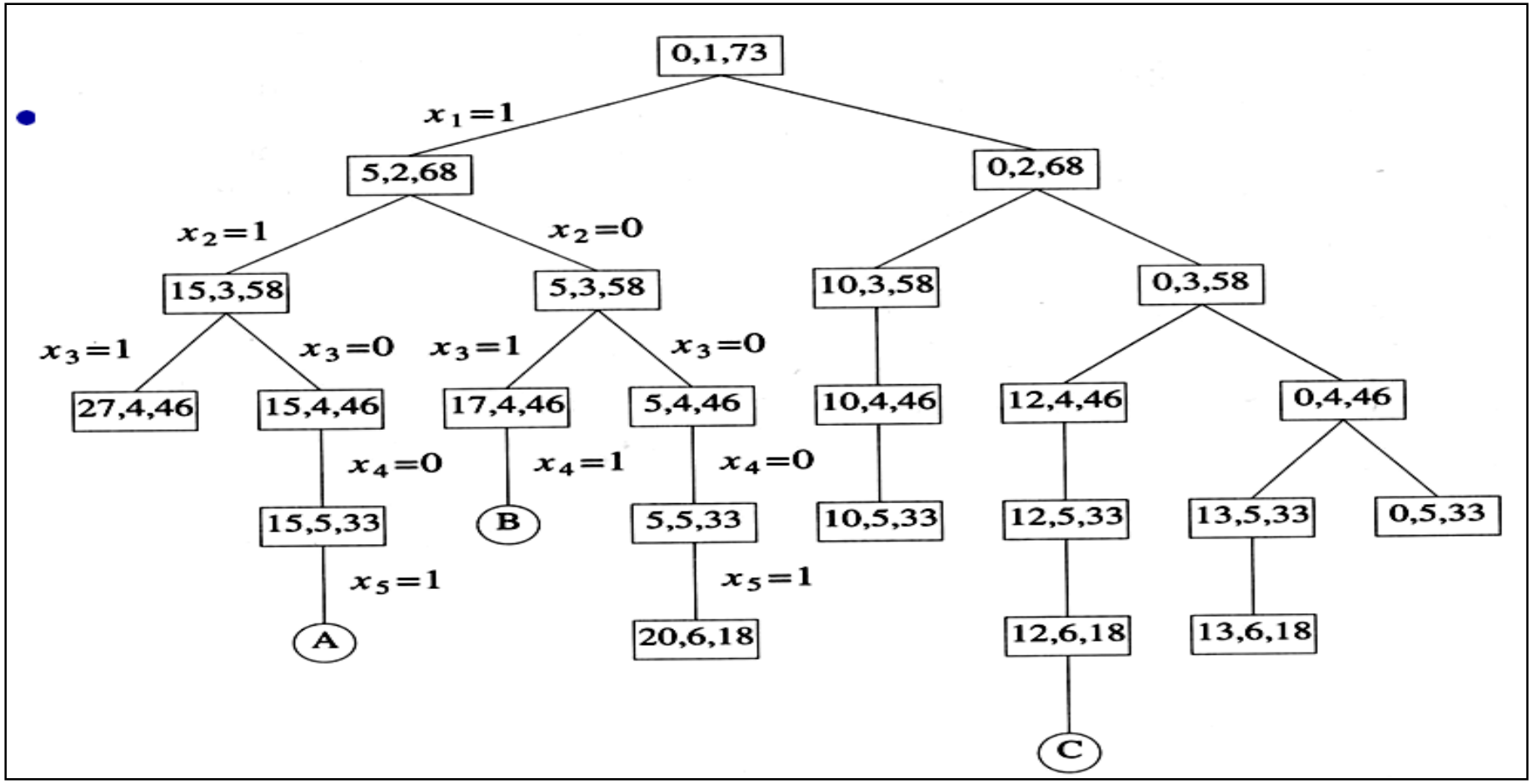
Example: consider a set S={5,10,12,13,15,18} and d=30. solve it for obtaining sum of subsets.

Initially subset-{}	Sum = 0	-
5	5 Since 5<30	Add next element
5,10	15 Since 15<30	Add next element
5,10,12	27 Since 27<30	Add next element
5,10,12,13	40 Since 40>30	Not feasible. Sum exceeds. Therefore Backtrack
5,10,12,15	42 Since 42>30	Not feasible. Sum exceeds. Therefore Backtrack
5,10,12,18	45 Since 45>30	Not feasible. Sum exceeds. Therefore Backtrack
5,10	15 Since 15<30	Add next element
5,10,13	28 Since 28<30	Add next element
5,10,13,15	43 Since 43>30	Not feasible. Sum exceeds. Therefore Backtrack
5,10,13,18	46 Since 46>30	Not feasible. Sum exceeds. Therefore Backtrack
5,10	15 Since 15<30	Add next element
5,10,15	30 Since 30=30	Solution obtained as sum=30=d

The state space tree is shown as below in figure. {5, 10, 12, 13, 15, 18}



We pass initially (0,1,73) to sum of sub sets. Here 0 represents sum, 1 represents index and 73 represents remaining sum (5+10+12+13+15+18)



Example: let m=31 and w={7,11,13,24}. Draw a portion of state space tree for solving sum of subset problem.

Algorithm SumOfSubset(s, k, r)

//Find all subsets of w[1:n] that sum to m. the value of x[j], $1 \le j \le k$, have already been determined. $S=\sum_{J=1}^{K-1} w[j]*x[j]$ and $r=\sum_{J=k}^{n} w[j]$. The w[j]'s are in non decreasing order. It is assumed that $w[1] \le m$ and $\sum_{i=1}^{n} w[i] > m$.

```
{     //Generate Left Child.  Note: s+w[k] ≤ m since B_{k-1} is true.
    x[k] := 1;
    if (S + w[k] = m) then
      write(x[1:k]);        //subset found and there is no recursive call here as w[j]>0, 1 ≤ j ≤ n
    else if (S + w[k] + w[k+1] ≤ m) then
       SumOfSubset(s + w[k], k+1, r - w[k]);
         //Generate right child and evaluate B_k.
    if(s + r - w[k] ≥ m) and (S+w[k+1] ≤ m) then
    {
      x[k]:=0;
      SumOfSubset(S, k+1, r - w[k]);
    }
}
```

5.2.3. Graph Coloring Problem

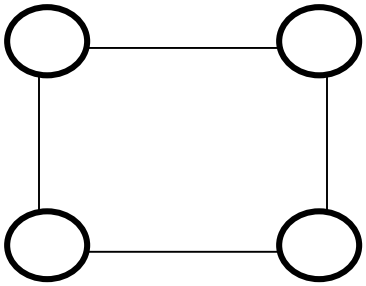
Let G is a graph and m be the given positive integer.

We want to discover whether 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. This is termed the m-colourability Decision Problem".

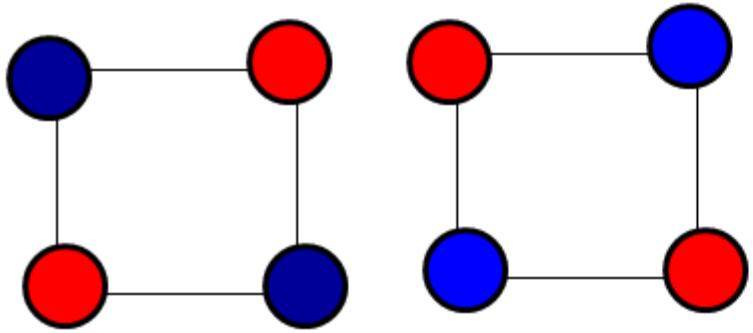
If d is the degree of the given graph, then it can be colored with d+1 colors.

The m-colourability optimization problem asks for the smallest integer m for which the graph G can be colored. This integer referred as the Chromatic Number of the graph.

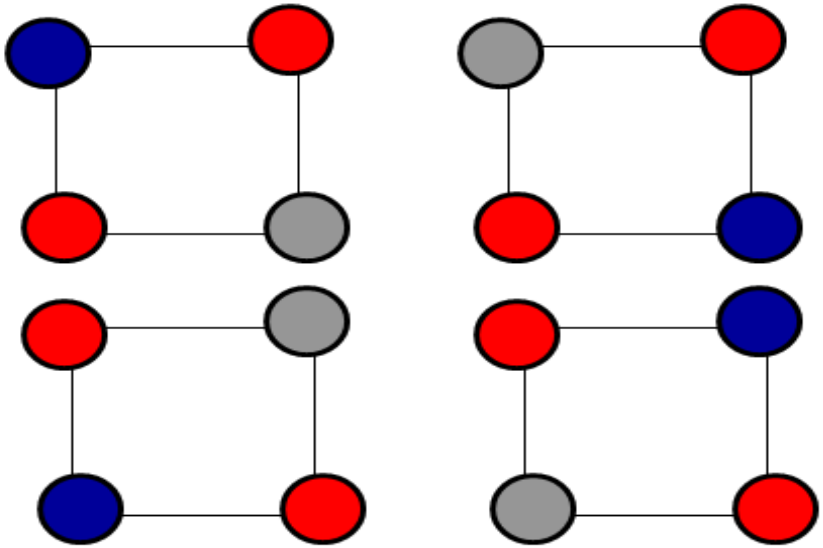
Example:



Number of Possible ways : 2



No.of Colors used: 2

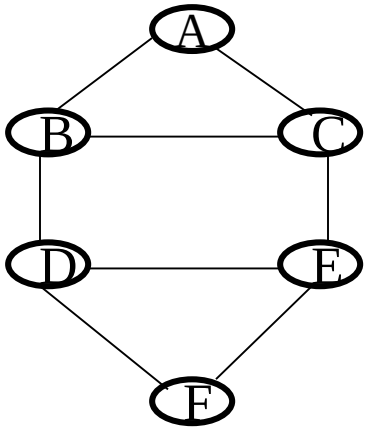


Some possible ways

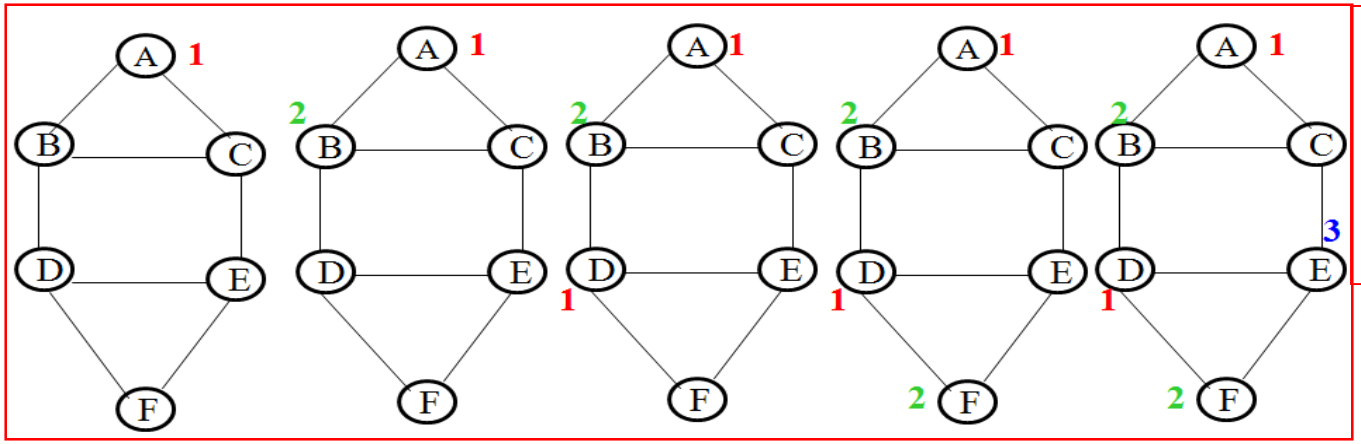
No.of Colors used: 3

A graph is said to be planar iff it can be drawn in a plane in such a way that no two edges cross each other

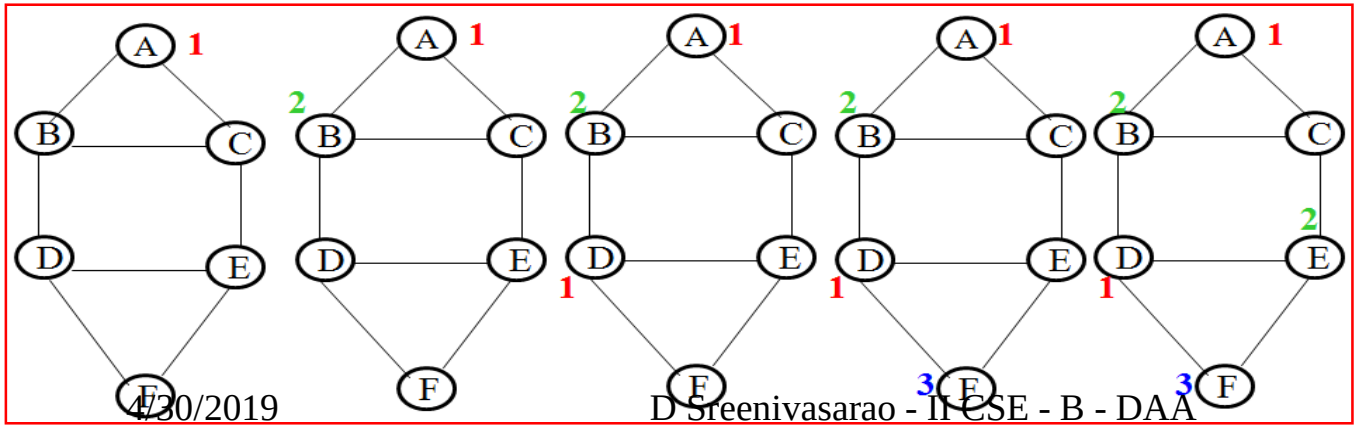
Example:

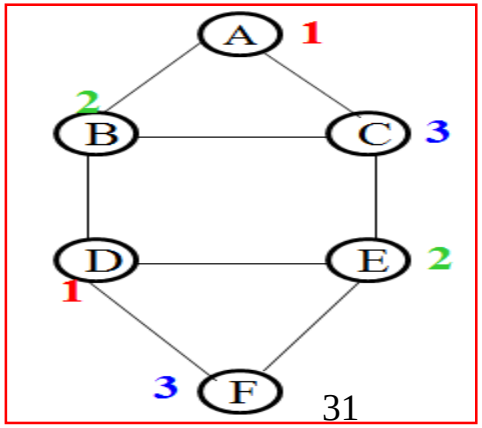


Step : A Graph G consists of vertices from A to F. there are three colors used RED, GREEN, BLUE. We will number them out. That means 1 indicates RED, 2 indicates GREEN and 3 indicates BLUE color.

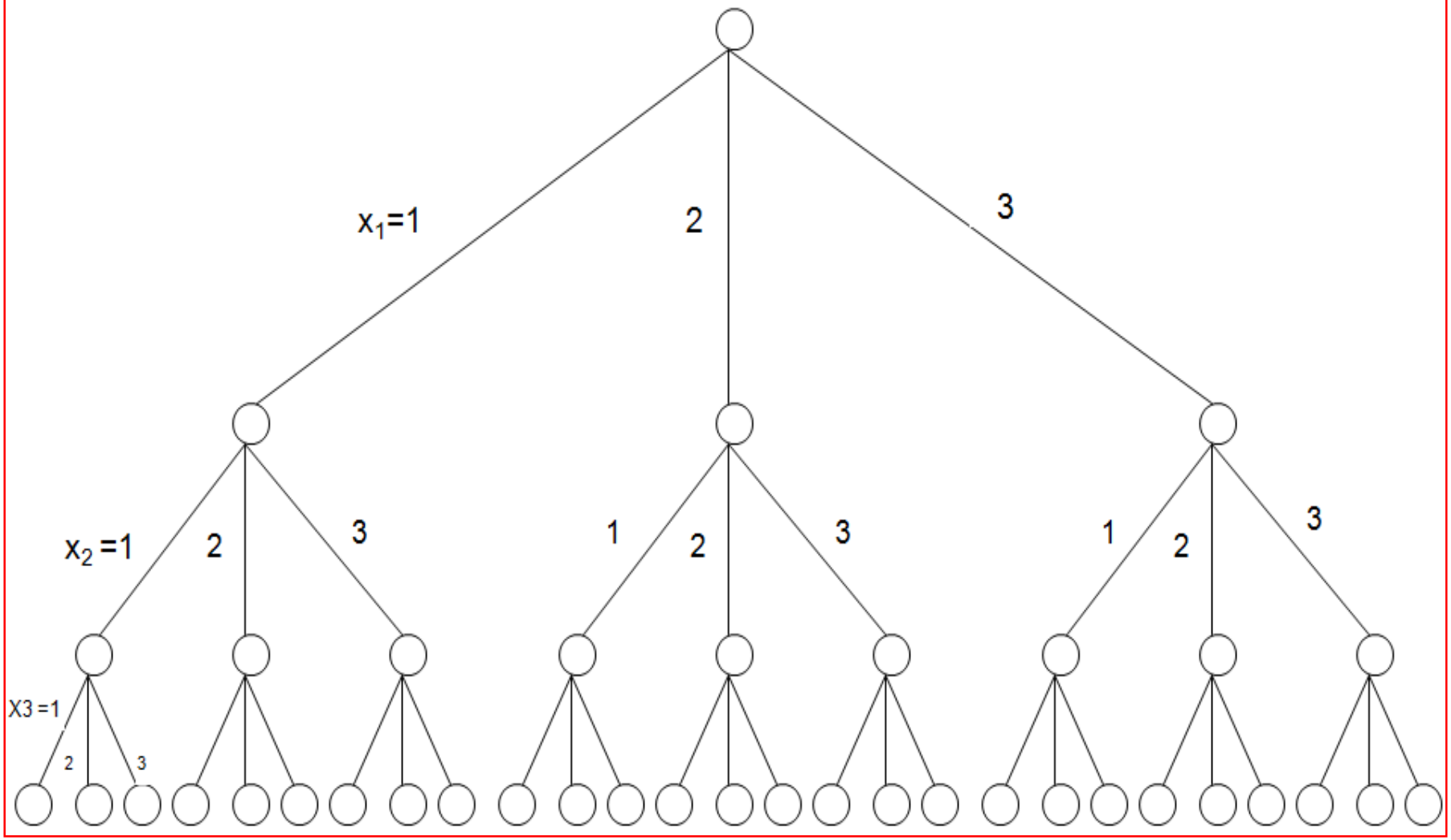


Stuck here!!
Can not assign 1 or 2 or 3. hence back track

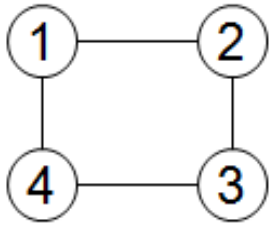


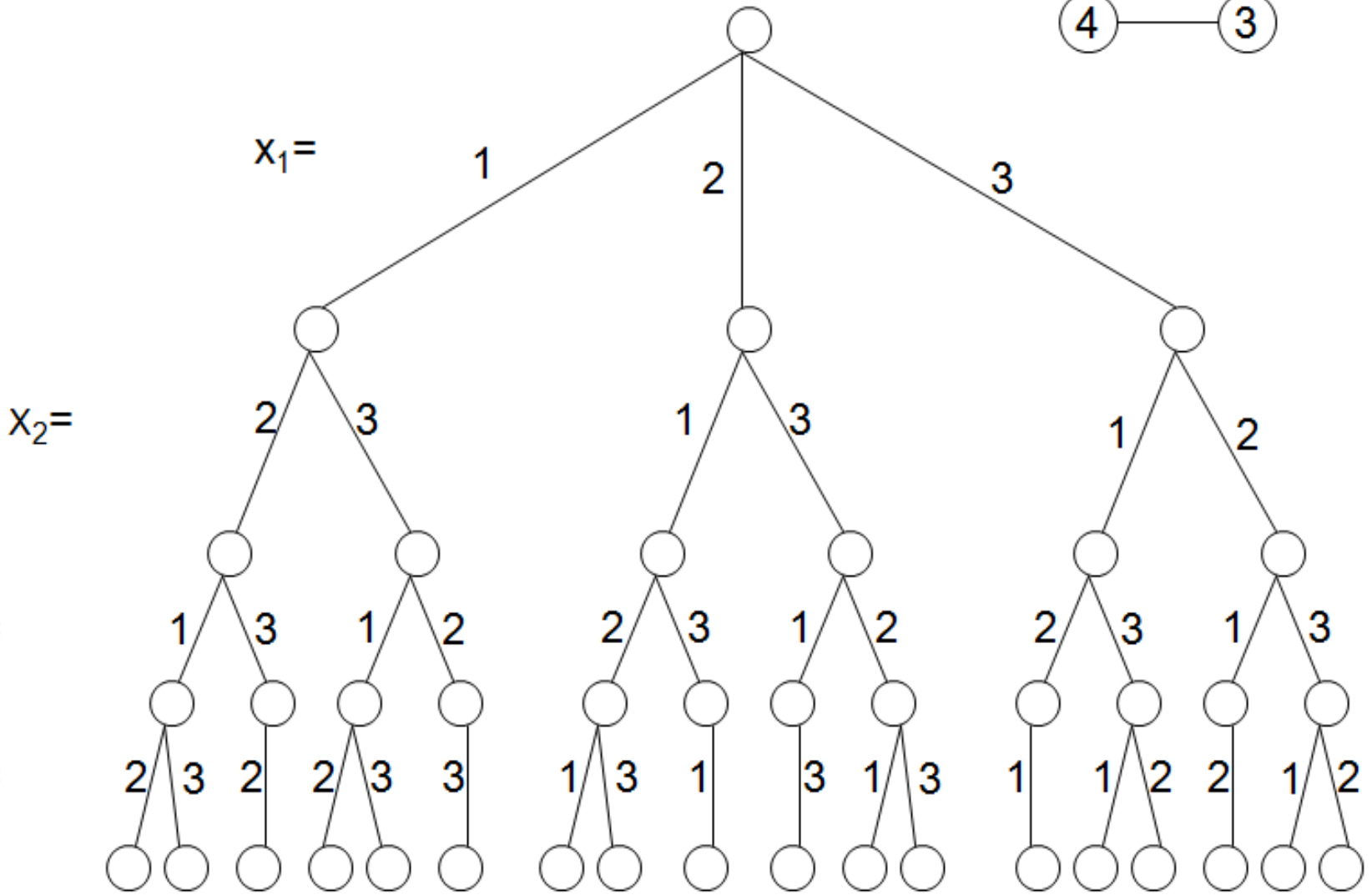


Solution space tree for mColoring when n=3 and m=3



A 4-node graph and all possible 3-colorings





```
Algorithm mColoring( k )
// this algorithm was formed using the recursive back tracking schema. 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 assignments 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     // At most 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 adjacency vertices have
//distinct integers. A value for x[k] is determined in the range [ 0,m ]. X[k] is assigned the next highest numbered
//color while maintaining distinctness from the adjacent vertices of vertex k. if no such color exists then x[k]=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
             if((G[k,j]≠0)and(x[k]=x[j]))then //if (k,j) is an edge if adj. vertices have the same color.
                 break;
          if( j = n+1 ) then // new Color found
              return;
   } until ( false );
}
```

Time Complexity:

At each internal node O(mn) time is spent by NEXTVALUE to determine the children corresponding to legal coloring.

Hence the total time is bounded by

$$\sum_{i=1}^{n} m^i.n = n(m + m^2 + m^3 + \ldots\ldots\ldots + m^n).$$

$$= n.\ m^n$$

$$= O(n.m^n)$$

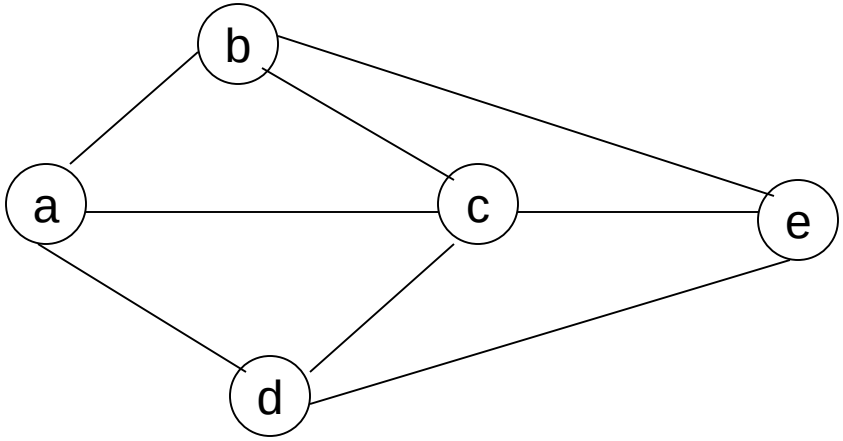
5.2.4.Hamiltonian Cycle:

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 and returns to its starting position suggested by William Hamilton.

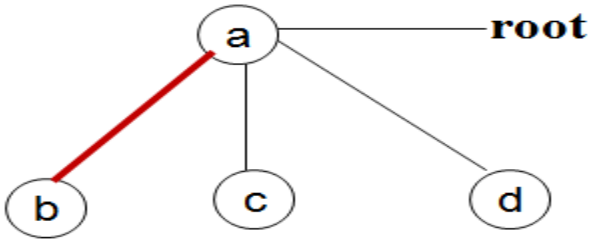
A circuit $x_0,x_1,\ldots.x_{n-1},x_n$, x0(with n>1) in a graph G=(V,E) is called Hamiltonian circuit if x_0, $x_1,\ldots.x_n$ is a Hamiltonian Path.

Example: Consider a graph G=(V,E), find a Hamiltonian circuit using back tracking method.

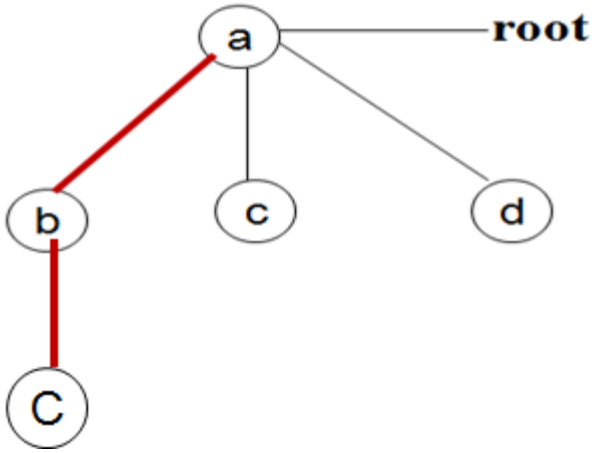


Step 1: initially we start our search with vertex ‘a’. The vertex ‘a’ becomes the of our implicit tree.

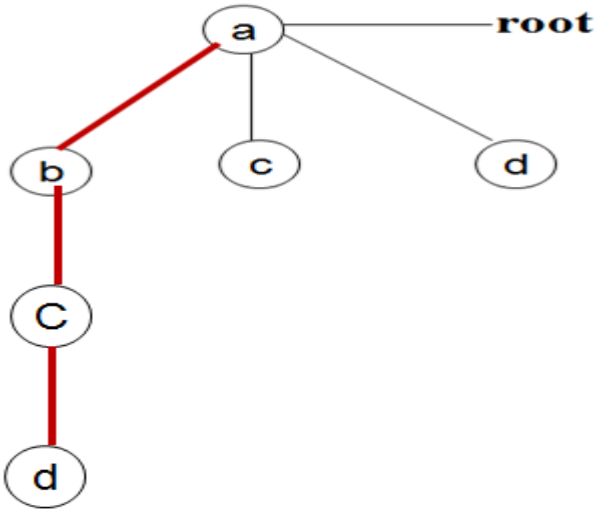
Next, we chose vertex ‘b’ adjacent to ‘a’ as it comes first in lexicographical order or (b,c,d)



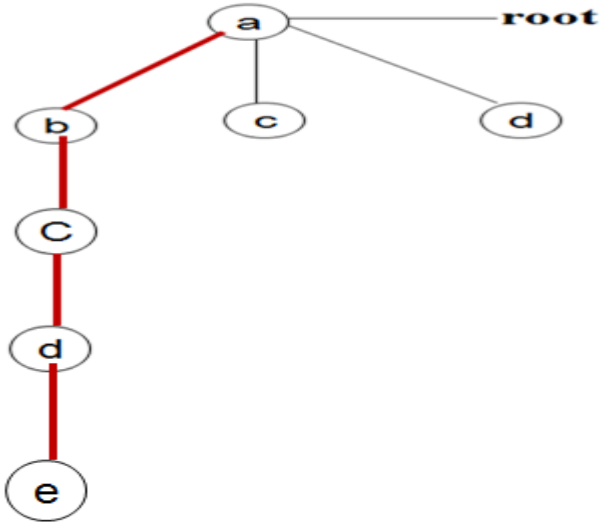
Next vertex ‘c’ is selected which is adjacent to ‘b’ and which comes first in lexicographical or (c,e)



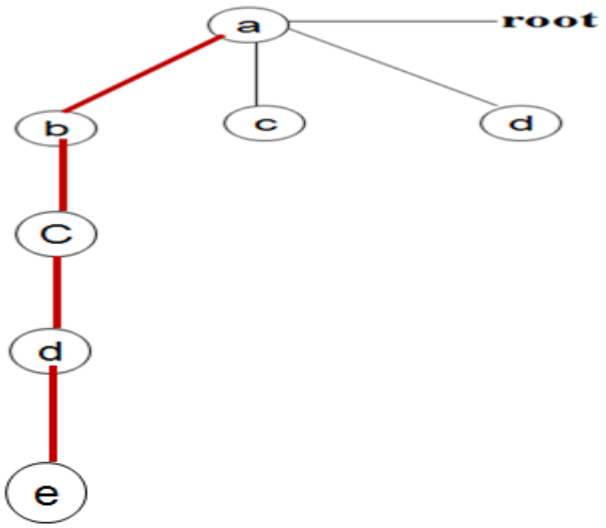
Next vertex ‘d’ is selected which is adjacent to ‘c’ and which comes first in lexicographical or (d,e)



Next vertex ‘e’ is selected. which is adjacent to ‘d’ and which comes first in lexicographical or (e)



Adjacent to ‘e’ are (b,c,d), but they are already visited. The vertex ‘a’ not visited directly from ‘e’. Thus we get the dead end. So we do not get the Hamiltonian cycle.



Dead end

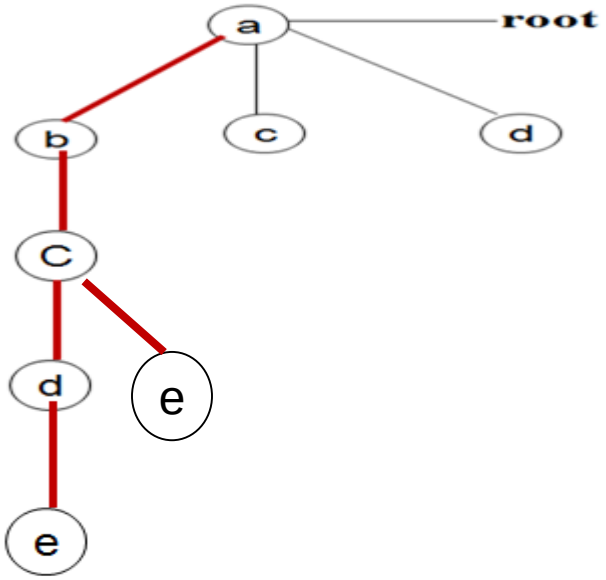
So, we back track one step and remove the vertex ‘e’ from our partial solution.

The vertex adjacent to ‘d’ are e,c,a. from which vertex ‘e’ has already been checked and c and a are already visited.

So, we again back track one step.

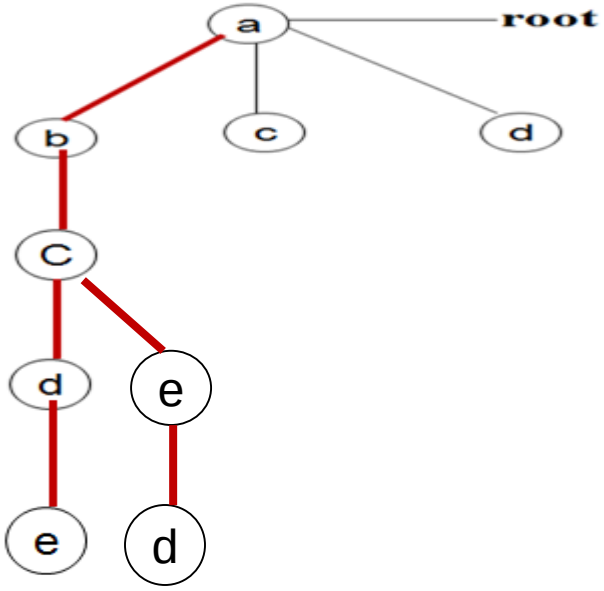
Hence we select the adjacent to c are b, d, e. the vertex b is already visited.

The vertex d already verified. So, now chose the vertex ‘e’ .



Dead end

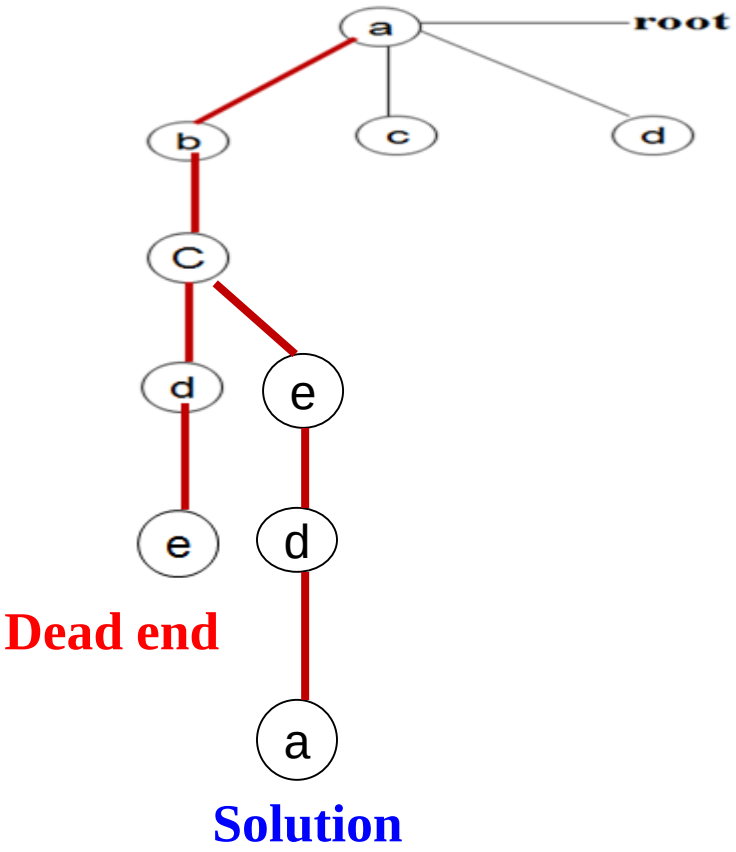
Adjacent to ‘e’ are (b, c, d), but b, c are already visited. The vertex ‘d’ is not visited . Thus we select ‘d’, which is adjacent to ‘e’.



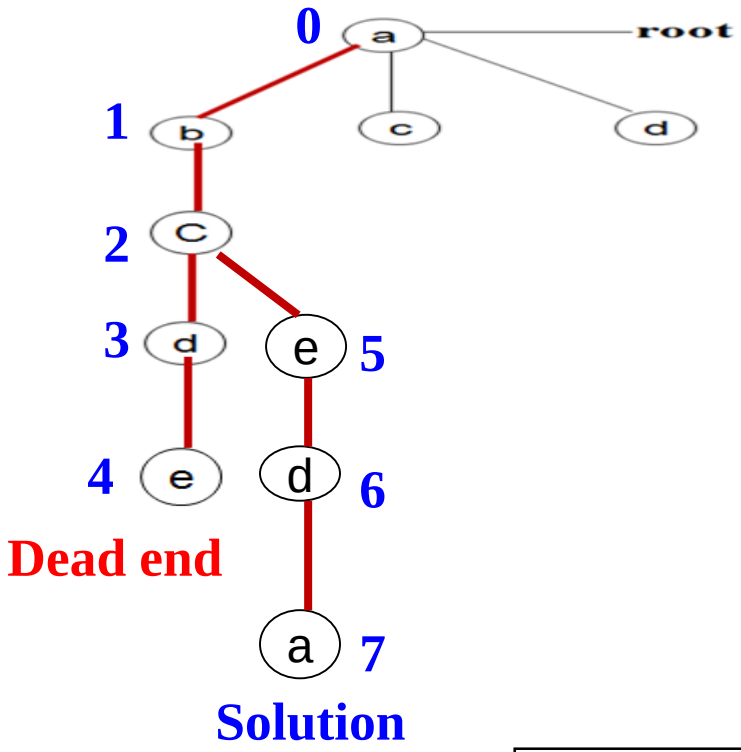
Dead end

Adjacent to ‘d’ are (a, c, e), all the vertices other than start vertex ‘a’ is visited only once.

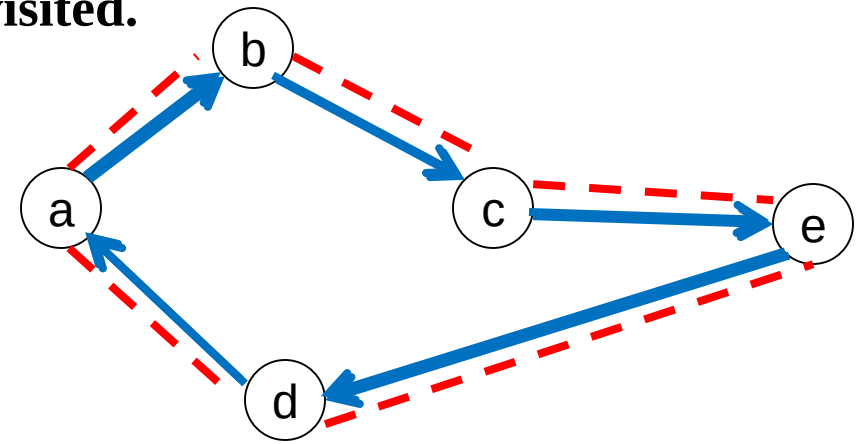
Hence we get the Hamiltonian cycle a-b-c-e-d-a.



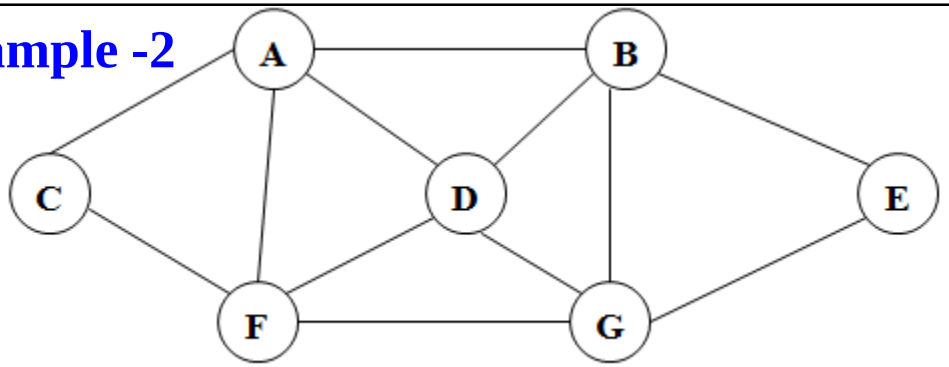
The final implicit tree for the Hamiltonian circuit is



The numbering of the node represents the order in which these nodes are visited.



Example -2



```
Algorithm Hamiltonian (k)
// this algorithm uses the recursive formulation of back tracking to find all the Hamiltonian cycles of a graph.
//The graph is stored as an adjacency matrix G[1:n,1:n]. All cycles being at node 1.
{
   repeat
    { //generate values for x[k]
       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 );
}
```

```
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 assigned to the next highest numbered vertex which does not already appear in x[k-1] and is
//connected by an edge to x[k-1], otherwise x[k]=0. if k=n then in addition x[k] is connected to x[1].
repeat
  {          // Generate values for x[k]
             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
          { // is there any edge
            for j := 1 to k-1 do
            if(x[j] = x[k]) then break; // check for distinctness
           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 );
}
```

Branch and Bound

- **Backtracking is effective for subset or permutation problems.**
- **Backtracking is not good for optimization problems. This drawback is rectified by using Branch And Bound technique.**
- **Branch And Bound is applicable for only optimization problems. Branch and Bound also uses bounding function similar to backtracking.**

5.3. Branch and Bound

- The term Branch and Bound 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.
- Both of BFS and D-search generalize to Branch and Bound strategies.
- In Branch and Bound terminology, a BFS-like space search will be called (First in First Out) search as the list of live nodes is a First In First Out List (or Queue)
- A D-Search 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 (or Stack)
- Three common search strategies are FIFO, LIFO and LC.
- The cost function $\hat{c}(.)$ such that $\hat{c}(x) \leq C(x)$ is used to provide lower bounds on solutions obtainable from any node x.
- If u is an upper bound on the cost of minimum cost solution then all live nodes x with $\hat{c}(x) > u$ may be killed as all answer nodes reachable from x have cost $C(x) \geq \hat{c}(x) > u$.
- In case an answer node with cost u has already been reached then all live nodes with $\hat{c}(x) \geq u$ may be killed.

We will use three types of search strategies or Nodes will be expanded in three ways.

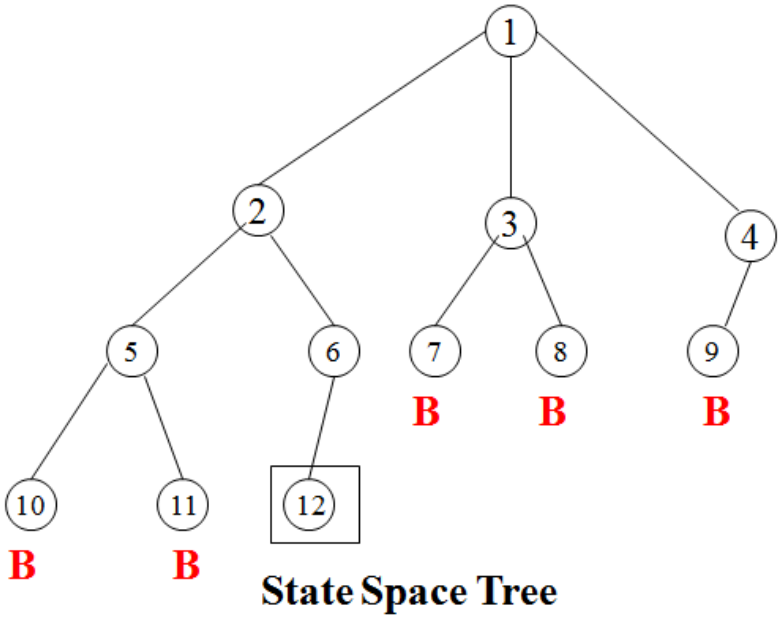
1. FIFO Branch and Bound Search – Queue

2. LIFO Branch and Bound Search or DFS – Stack

Least-Cost (or max priority) Branch and Bound Search — Priority Queue.

Least-cost branch and bound directs the search to the parts which most likely to contain the answer.

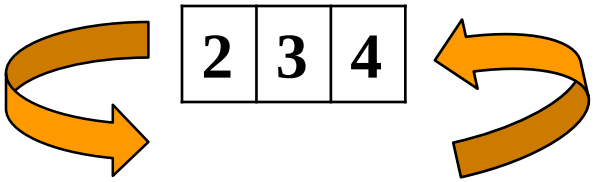
1. FIFO Branch And Bound Algorithm



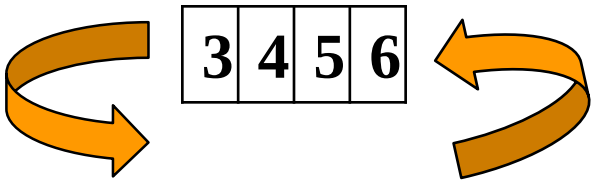
State Space Tree

Assume that the node 12 is an answer node(solution).

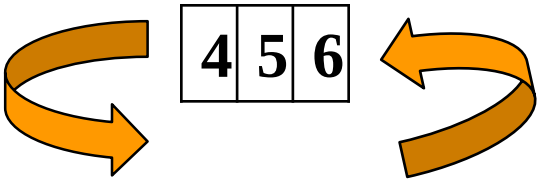
In FIFO search, first we will take E-node as 1. next, we generate the children of node 1. we will place all these live nodes in a Queue.



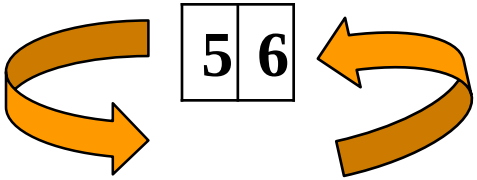
Now we delete an element from Queue, that is node 2. Next generate children of node 2 and place in the Queue



Next delete an element from queue and take it as E-node, generate the children of node 3. 7 and 8 are children of 3 and these live nodes are killed by bounding function. So we will not include in the queue.



Again delete an element from queue. Take as E-node. Generate the children of 4. node 9 is generated and killed by bounding function.



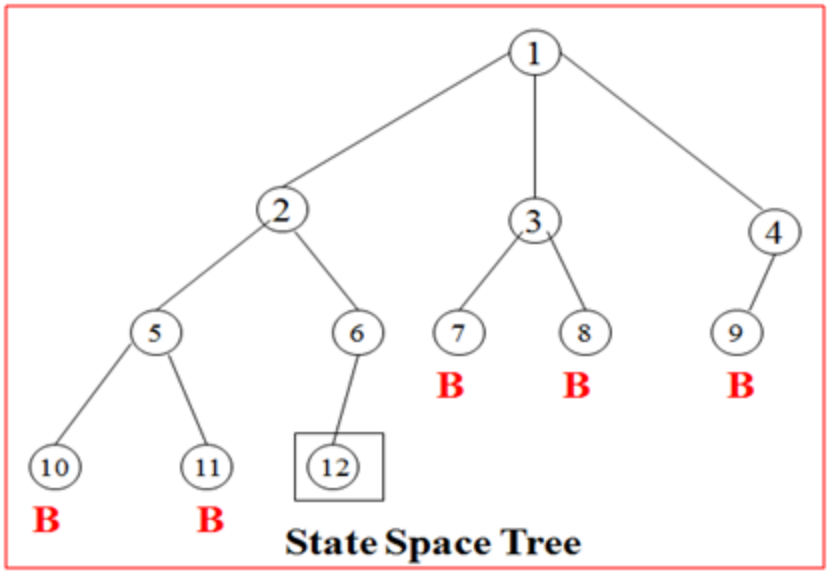
Next delete an element from queue. Generate children of node 5. That is nodes 10 and 11 are generated and killed by bounding function.

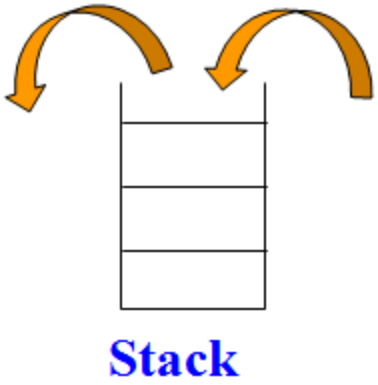
Last node in queue is 6. the child of node 6 is 12 and it satisfies the conditions of the problem. Which is the answer node. So search terminates.

2. LIFO Branch-and-bound:

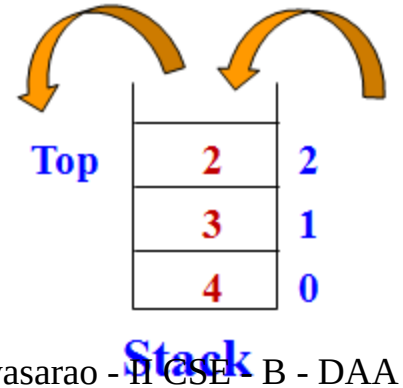
For this we will use a data structure called Stack.

Initially stack is empty.

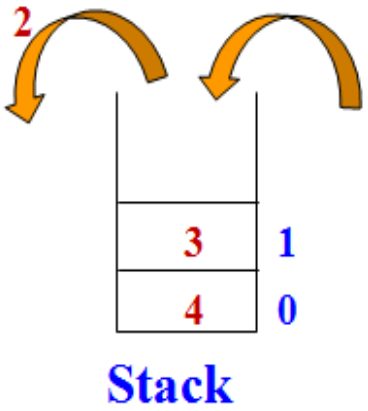




Generate children of node 1 and place these live nodes into stack.

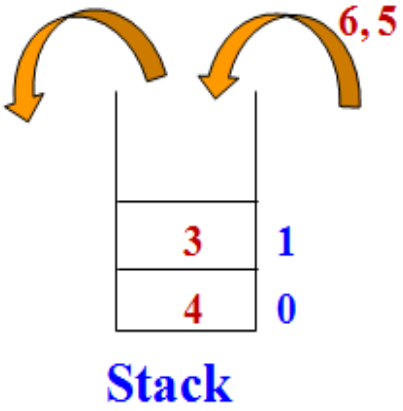


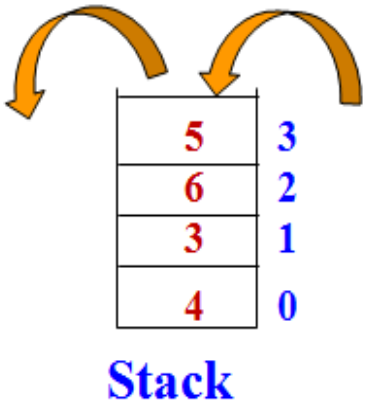
Remove from the stack and generate the children of it. Place these nodes in to stack.



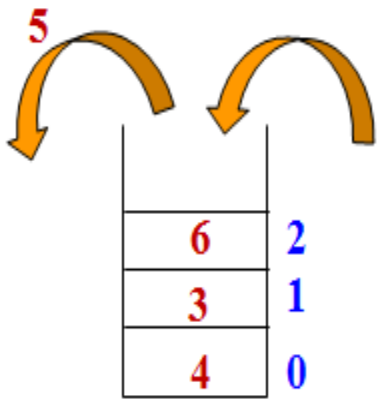
2 is removed from the stack. The children of 2 are 5 and 6.

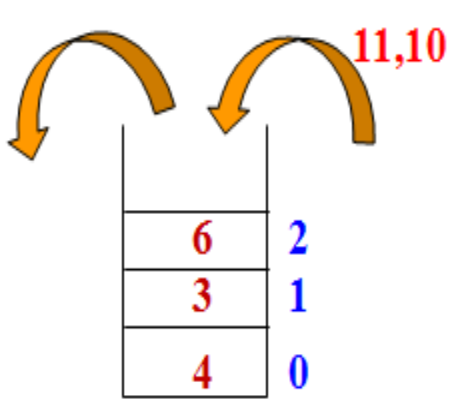
Now the contents of stack are

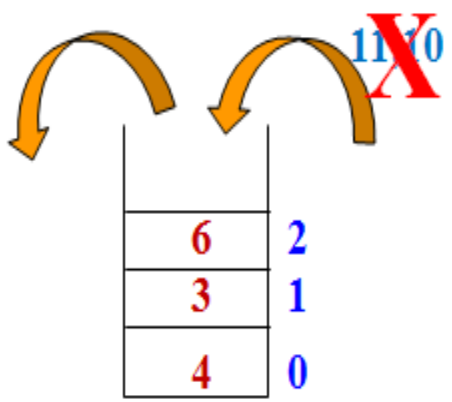




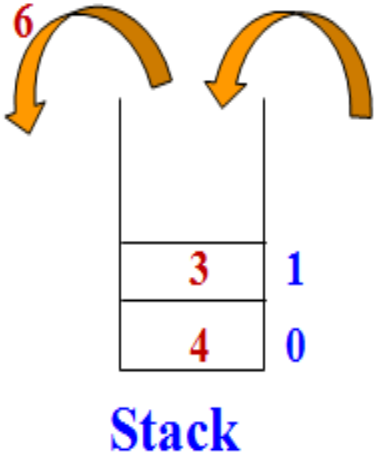
Again remove an element from stack, that is, node 5 is removed and nodes generated by 5 are 10,11 which are killed by bounding function. So, we will not place 10,11 into stack.

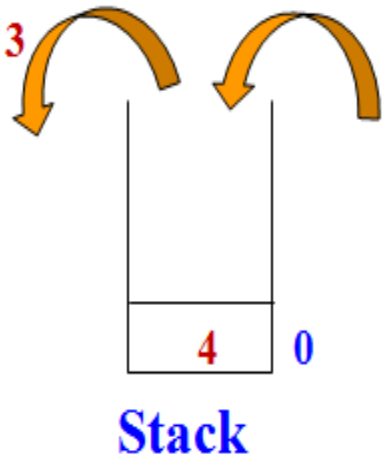


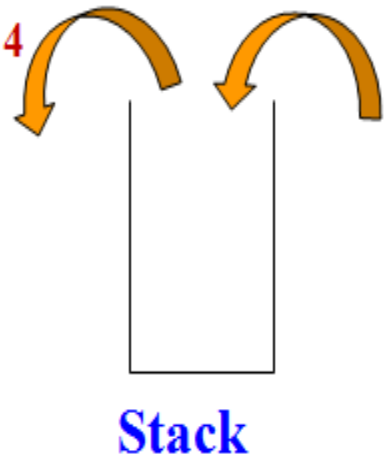




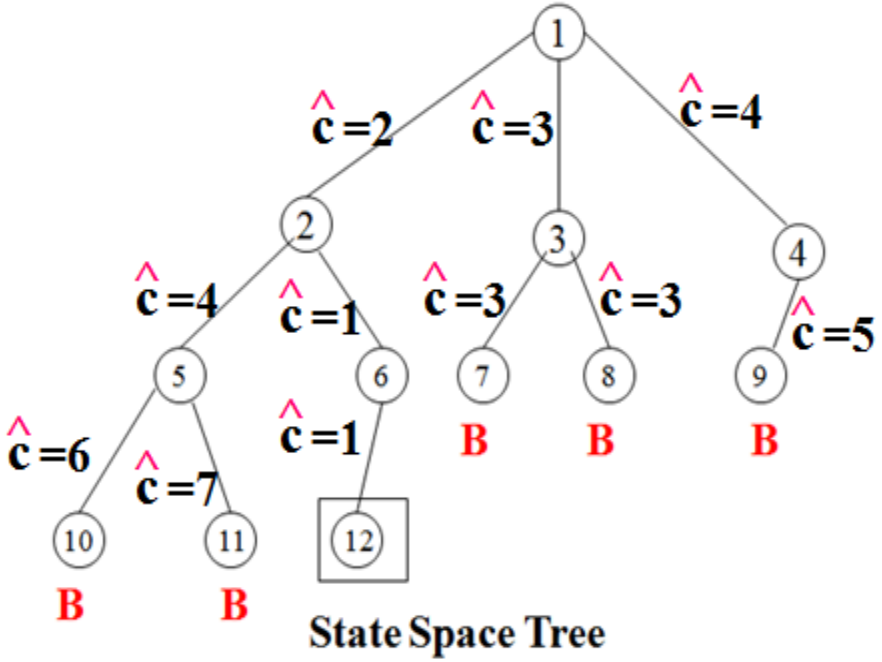
Generate node 6, that is 12. it is the answer node. So, search process terminates.







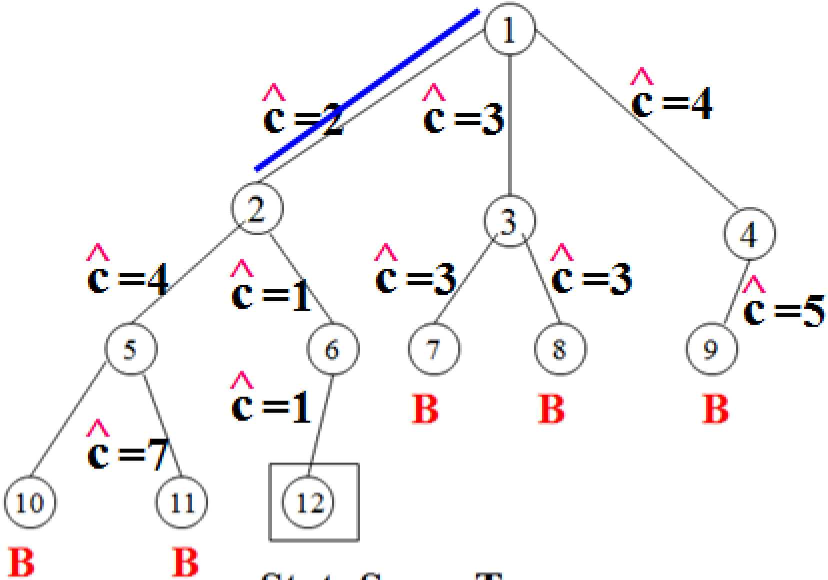
3. Least Cost branch and Bound (LCBB)



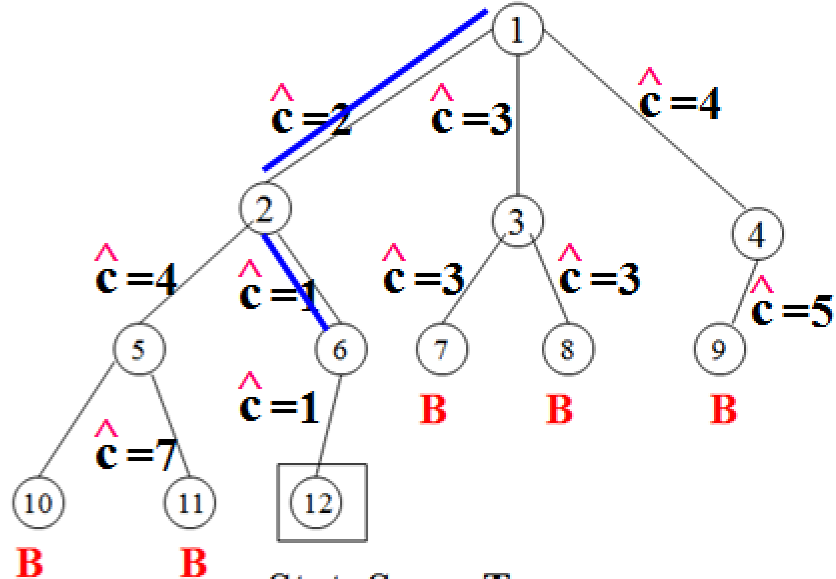
State Space Tree

Generally, children of node 1 are 2,3,4. by using Ranking Function we will calculate the cost of 2,3 and 4 nodes is 2,3 and 4 respectively.

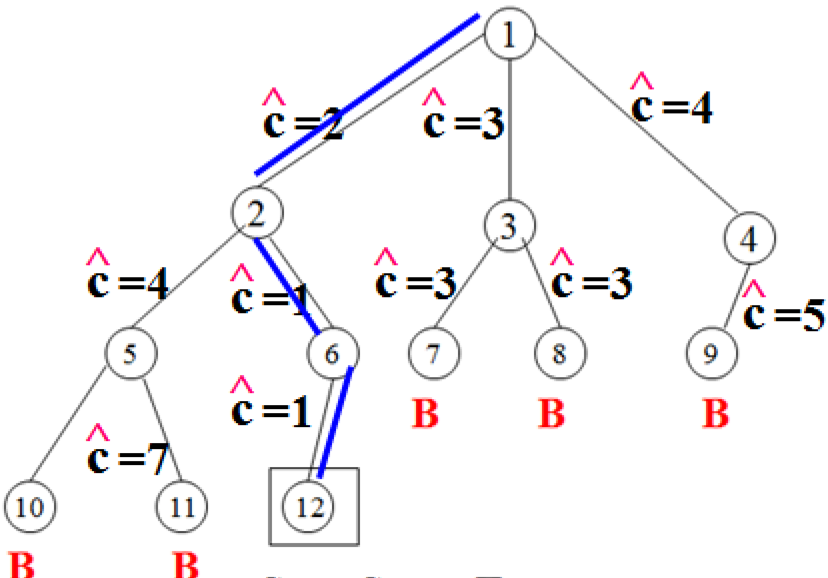
Now we will select a node which has least cost, that is node 2. for node 2, the children are 5 and 6. the least cost among 5 and 6 are 1, that is node 6.



State Space Tree



State Space Tree



State Space Tree

Now we will generate the children of node 6., that is 12. we will select least cost 12 node. So, 1 is the least cost and the node is 12. moreover the node 12 is the answer node. So, terminate the search process.

5.3.1. Branch – and - Bound Control Abstraction:

Let t be a state space tree and c(.) is a cost function for the nodes. If x is node in t, then c(x) is the minimum cost of any answer node in the sub tree with root x.

C(t) is the cost of minimum cost answer node in t. usually it is not possible to find an easily computable function c(.). An estimate $\hat{c}$ of c() is used. It is easy to compute $\hat{c}$ has the property that if x either an answer node or leaf node then $c(x) = \hat{c}(x)$.

A LCBB search of tree uses two functions least() and add(). Begin with an upper bound with and node 1 as the first node. When node 1 is expanded, node 2,3,4,…. are generated in the order. Add the live node to the list of live nodes. If the function least() finds least cost node and the node is deleted from the list of live nodes and returned. The output of least cost search is tracing the path from the answer node to root node of the tree.

```
Algorithm LC_Search (t)    // search tree t for an answer node
{
    if *t is an answer node then output *t and return;
    E=t;     // E-node
    Initialize the list of live nodes to be empty;
    repeat
    {
        for (each child x of E) do
        {
            if (x is an answer node) then output the path from x to t and return;
            Add(x);   // x is a new live node
            (x->parent):=E;    // pointer for path to root node
        }
        if (there are no more live nodes) then
        {
            write("No answer node");
            return;
        }
        E=Least();
    } until (false);
}
```

```
list node = record
{
    list node *next,*parent;
    float cost;
}
```

(x ->parent)=E is to print answer path.

1. It is desirable to find an answer node that has minimum cost among all answer nodes. But LC search can not guarantee to find an answer node G with minimum cost c(G).
2. When there exists two nodes in a graph such that c(x)>c(y) in one branch while c(x) < c(y) in other branch, LC search can not find minimum cost answer node.
3. Even if c(x) < c(y) for every pair of nodes x, y such that c(x)<c(y), LC may not find a minimum cost answer node.
4. When the estimate $\hat{c}()$ for a node is less than the cost c() then a slight modification to the LC search results in search algorithm that terminates when a minimum cost answer node is reached. In this modification, the search continues until an answer node becomes E-node.
5. At the time an E-node is an answer node $\hat{c}(E) \leq \hat{c}(L)$ for every node on a graph L in the list of live nodes. Hence $\hat{c}(E) \leq \hat{c}(L)$ and so E is minimum cost answer node.

5. 4. The Branch and Bound Applications are

5.4.1. Travelling Sales Person Problem

5.4.2. 0/1 Knapsack Problem-LCBB.

5.4.3. 0/1 Knapsack Problem-FIFOBB.

5.4.1. Travelling Sales Person Problem

Problem Statement : If there are n cities and cost of traveling from any city to any other city is given. Then we have to obtain the shortest round-trip such that each city is visited exactly once and then returning to starting city, completes the tour.

Procedure for solving traveling sale person problem:

1. Reduce the given cost matrix. A matrix is reduced if every row and column is reduced. A row (column) is said to be reduced if it contain at least one zero and all-remaining entries are non-negative. This can be done as follows:

   a) *Row reduction:* Take the minimum element from first row, subtract it from all elements of first row, next take minimum element from the second row and subtract it from second row. Similarly apply the same procedure for all rows.

   b) Find the sum of elements, which were subtracted from rows.

   c) Apply column reductions for the matrix obtained after row reduction.

      Column reduction: Take the minimum element from first column, subtract it from all elements of first column, next take minimum element from the second column and subtract it from second column. Similarly apply the same procedure for all columns.

   d) Find the sum of elements, which were subtracted from columns.

   e) Obtain the cumulative sum of row wise reduction and column wise reduction.

      Cumulative reduced sum = Row wise reduction sum + column wise reduction sum.

      Associate the cumulative reduced sum to the starting state as lower bound and ∞ as upper bound.

2. Calculate the reduced cost matrix for every node R. Let A is the reduced cost matrix for node R. Let S be a child of R such that the tree edge (R, S) corresponds to including edge <i, j> in the tour. If S is not a leaf node, then the reduced cost matrix for S may be obtained as follows:

   a) Change all entries in row i and column j of A to ∞.

   b) Set A (j, 1) to ∞.

   c) Reduce all rows and columns in the resulting matrix except for rows and column containing only ∞. Let r is the total amount subtracted to reduce the matrix.

   c) Find $c(S) = c(R) + A(i, j) + r$, where 'r' is the total amount subtracted to reduce the matrix, $c(R)$ indicates the lower bound of the i^{th} node in (i, j) path and $c(S)$ is called the cost function.

3. Repeat step 2 until all nodes are visited.

Solve the following instance of Travelling Sales Person Problem using LCBB.

The cost matrix is

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

Solution:

Step 1: Find the reduced cost matrix.

Apply row reduction method:

Deduct 10 (which is the minimum) from all values in the 1st row.
Deduct 2 (which is the minimum) from all values in the 2nd row.
Deduct 2 (which is the minimum) from all values in the 3rd row.
Deduct 3 (which is the minimum) from all values in the 4th row.
Deduct 4 (which is the minimum) from all values in the 5th row.

The resulting row wise reduced cost matrix =

$$\begin{bmatrix} \infty & 10 & 20 & 0 & 1 \\ 13 & \infty & 14 & 2 & 0 \\ 1 & 3 & \infty & 0 & 0 \\ 16 & 3 & 15 & \infty & 0 \\ 12 & 0 & 3 & 12 & \infty \end{bmatrix}$$

Row wise reduction sum = 10 + 2 + 2 + 3 + 4 = 21

Now apply column reduction for the above matrix:

Deduct 1 (which is the minimum) from all values in the 1st column.
Deduct 3 (which is the minimum) from all values in the 3rd column.

The resulting column wise reduced cost matrix (A) =

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

Column wise reduction sum = 1 + 0 + 3 + 0 + 0 = 4

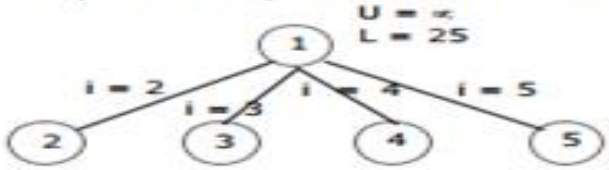
Cumulative reduced sum = row wise reduction + column wise reduction sum.
= 21 + 4 = 25.

This is the cost of a root i.e., node 1, because this is the initially reduced cost matrix.

The lower bound for node 1 is 25 and upper bound is ∞.

Starting from node 1, we can next visit 2, 3, 4 and 5 vertices. So, consider to explore the paths (1, 2), (1, 3), (1, 4) and (1, 5).

The tree organization up to this point is as follows:



Variable 'i' indicates the next node to visit.

Step 2:

Consider the path (1, 2):

Change all entries of row 1 and column 2 of A to ∝ and also set A(2, 1) to ∝.

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

Apply row and column reduction for the rows and columns whose rows and columns are not completely ∝.

Then the resultant matrix is

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

Row reduction sum = 0 + 0 + 0 + 0 = 0
Column reduction sum = 0 + 0 + 0 + 0 = 0
Cumulative reduction (r) = 0 + 0 = 0

Therefore, as $c(S) = c(R) + A(1, 2) + r$

$c(S) = 25 + 10 + 0 = 35$

Consider the path (1, 3):

Change all entries of row 1 and column 3 of A to ∞ and also set A(3, 1) to ∞.

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

Apply row and column reduction for the rows and columns whose rows and columns are not completely ∞.

Then the resultant matrix is $\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & \infty & 2 & 0 \\ \infty & 3 & \infty & 0 & 2 \\ 4 & 3 & \infty & \infty & 0 \\ 0 & 0 & \infty & 12 & \infty \end{bmatrix}$

Row reduction sum = 0
Column reduction sum = 11
Cumulative reduction (r) = 0 + 11 = 11

Therefore, as $c(S) = c(R) + A(1,3) + r$

$c(S) = 25 + 17 + 11 = 53$

Consider the path (1, 4):

Change all entries of row 1 and column 4 of A to ∞ and also set A(4, 1) to ∞.

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

Apply row and column reduction for the rows and columns whose rows and columns are not completely ∞.

Then the resultant matrix is $\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ \infty & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & \infty & \infty \end{bmatrix}$

Row reduction sum = 0
Column reduction sum = 0
Cumulative reduction (r) = 0 + 0 = 0

Therefore, as $c(S) = c(R) + A(1,4) + r$

$c(S) = 25 + 0 + 0 = 25$

Consider the path (1, 5):

Change all entries of row 1 and column 5 of A to ∞ and also set A(5, 1) to ∞.

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

Apply row and column reduction for the rows and columns whose rows and columns are not completely ∞.

Then the resultant matrix is

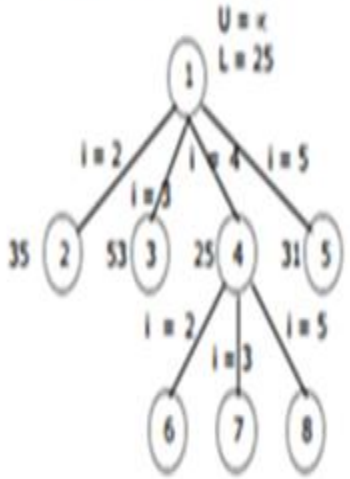
$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 10 & \infty & 9 & 0 & \infty \\ 0 & 3 & \infty & 0 & \infty \\ 12 & 0 & 9 & \infty & \infty \\ \infty & 0 & 0 & 12 & \infty \end{bmatrix}$$

Row reduction sum = 5
Column reduction sum = 0
Cumulative reduction (r) = 5 + 0 = 0

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

$\hat{c}(S)$ = 25 + 1 + 5 = 31

The tree organization up to this point is as follows:



The cost of the paths between (1, 2) = 35, (1, 3) = 53, (1, 4) = 25 and (1, 5) = 31. The cost of the path between (1, 4) is minimum. Hence the matrix obtained for path (1, 4) is considered as reduced cost matrix.

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

The new possible paths are (4, 2), (4, 3) and (4, 5).

Consider the path (4, 2):

Change all entries of row 4 and column 2 of A to ∞ and also set A(2, 1) to ∞.

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

Apply row and column reduction for the rows and columns whose rows and columns are not completely ∞.

Then the resultant matrix is

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

Row reduction sum = 0
Column reduction sum = 0
Cumulative reduction (r) = 0 + 0 = 0

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

$\hat{c}(S) = 25 + 3 + 0 = 28$

Consider the path (4, 3):

Change all entries of row 4 and column 3 of A to ∞ and also set A(3, 1) to ∞.

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

Apply row and column reduction for the rows and columns whose rows and columns are not completely ∞.

Then the resultant matrix is

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

Row reduction sum = 2
Column reduction sum = 11
Cumulative reduction (r) = 2 + 11 = 13

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

$\hat{c}(S) = 25 + 12 + 13 = 50$

Consider the path (4, 5):

Change all entries of row 4 and column 5 of A to ∞ and also set A(5, 1) to ∞.

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

Apply row and column reduction for the rows and columns whose rows and columns are not completely ∞.

Then the resultant matrix is

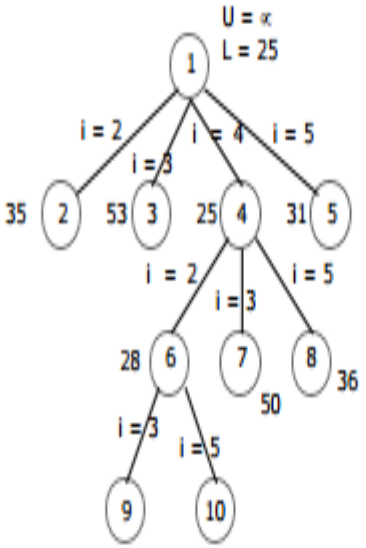
$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & 0 & \infty & \infty \\ 0 & 3 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & 0 & \infty & \infty \end{bmatrix}$$

Row reduction sum = 11
Column reduction sum = 0
Cumulative reduction (r) = 11+0 = 11

Therefore, as $\hat{c}(S) = \hat{c}(R) + A(4,5) + r$

$\hat{c}(S)$ = 25 + 0 + 11 = 36

The tree organization up to this point is as follows:



The cost of the paths between (4, 2) = 28, (4, 3) = 50 and (4, 5) = 36. The cost of the path between (4, 2) is minimum. Hence the matrix obtained for path (4, 2) is considered as reduced cost matrix.

$$A = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & \infty & 0 \\ 0 & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & 0 & \infty & \infty \end{bmatrix}$$

The new possible paths are (2, 3) and (2, 5).

Consider the path (2, 3):

Change all entries of row 2 and column 3 of A to ∞ and also set A(3, 1) to ∞.

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

Apply row and column reduction for the rows and columns whose rows and columns are not completely ∞.

Then the resultant matrix is $\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \end{bmatrix}$

Row reduction sum = 2
Column reduction sum = 11
Cumulative reduction (r) = 2 + 11 = 13

Therefore, as $\hat{c}(S) = \hat{c}(R) + A(2,3) + r$

$\hat{c}(S)$ = 28 + 11 + 13 = 52

Consider the path (2, 5):

Change all entries of row 2 and column 5 of A to ∞ and also set A(5, 1) to ∞.

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

Apply row and column reduction for the rows and columns whose rows and columns are not completely ∞.

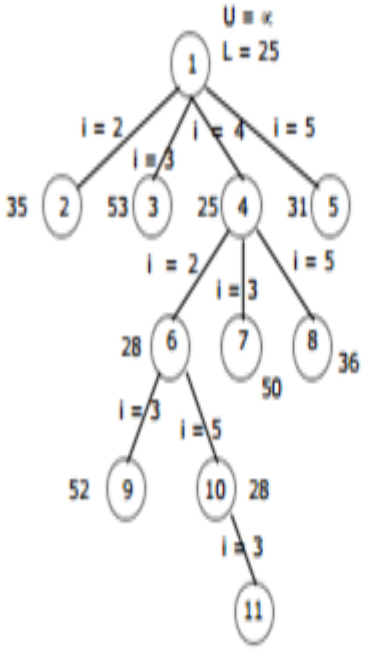
Then the resultant matrix is $\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty \end{bmatrix}$

Row reduction sum = 0
Column reduction sum = 0
Cumulative reduction (r) = 0 + 0 = 0

Therefore, as $\hat{c}(S) = \hat{c}(R) + A(2,5) + r$

$\hat{c}(S)$ = 28 + 0 + 0 = 28

The tree organization up to this point is as follows:



The cost of the paths between (2, 3) = 52 and (2, 5) = 28. The cost of the path between (2, 5) is minimum. Hence the matrix obtained for path (2, 5) is considered as reduced cost matrix.

$$A = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty \end{bmatrix}$$

The new possible paths is (5, 3).

Consider the path (5, 3):

Change all entries of row 5 and column 3 of A to ∞ and also set A(3, 1) to ∞. Apply row and column reduction for the rows and columns whose rows and columns are not completely ∞.

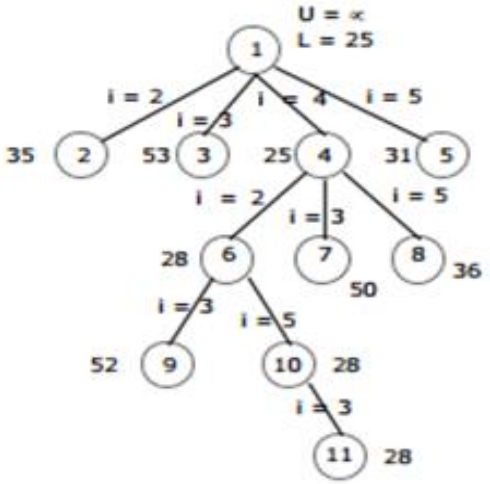
Then the resultant matrix is $\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \end{bmatrix}$

Row reduction sum = 0
Column reduction sum = 0
Cumulative reduction (r) = 0 + 0 = 0

Therefore, as $\hat{c}(S) = \hat{c}(R) + A(5, 3) + r$

$\hat{c}(S)$ = 28 + 0 + 0 = 28

The overall tree organization is as follows:



The path of traveling sale person problem is:

1 → 4 → 2 → 5 → 3 → 1

The minimum cost of the path is: 10 + 6 +2+ 7 + 3 = 28.

Traveling Sale Person Problem:

By using dynamic programming algorithm we can solve the problem with time complexity of $O(n^2 2^n)$ for worst case. This can be solved by branch and bound technique using efficient bounding function. The time complexity of traveling sale person problem using LC branch and bound is $O(n^2 2^n)$ which shows that there is no change or reduction of complexity than previous method.

0/1 Knapsack Problem

- **Generally Branch and Bound will minimize the objective function.**
- **The 0/1 knapsack problem is a maximization problem.**
- **This difficulty can be avoided by replacing the objective function $\sum p_i x_i$ by $-\sum p_i x_i$.**

Note: All live nodes with c(x) > upper can be killed when they are about to become E-nodes.

FIFO Branch and Bound-Steps to be followed in FIFOBB

Step 1:Compute c(.) and u(.) for a generated node, then if c>upper(initially upper = u(root node)) immediately kill the node otherwise add the node to the list of live nodes.

Step 2:Update upper = u(.), if upper is greater than u(.).

Step 3:The next node in the queue (FIFO) will be the next E-node.

Step 4:Whenever a node about to become an E-node, check if c(.) is greater than upper, if yes kill the node otherwise generate its children. Also kill the nodes which are not feasible.

Step 5: Continue step 1 to 4 till all the live nodes are expanded.

Step 6 : Finally the highest u value (absolute value) node will be the answer node. Trace the path in backward direction from answer node to root node for solution subset.

Example:- Draw the portion of state space tree generated by LCBB by the following Knapsack Problem. n=5, $(p_1,p_2,p_3,p_4,p_5)=(10,15,6,8,4)$ $(w_1,w_2,w_3,w_4,w_5)=(4,6,3,4,2)$ and m=12

Solution: First convert the profits to negative

$(p_1,p_2,p_3,p_4,p_5)=(-10,-15,-6,-8,-4)$

Calculate the lower bound and upper bound for each node

Place the first item in the bag. That is $w_1 = 4$

Remaining weight is 12 – 4 = 08.

Now, Place the 2^{nd} item in the bag. That is $w_2 = 6$

Remaining weight is 08 – 6 = 02.

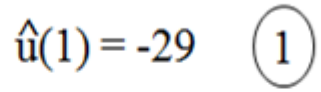
Since fractions are not allowed in calculation of upper bound, so we can not place the third and 4^{th} items.

Place 5^{th} item. There fore profits earned = -10 -15 - 4 = -29 = Upper bound

To calculate the lower bound, place the third item in the bag. Since fractions are allowed.

Lower bound $\hat{u}$ = -10 -15-(2/3)*6 = -29

For Node (1) $\hat{u}(1) = -29$ $\hat{c}(1) = -29$



For node 2: ($x_1 = 1$ means we should place first item in the bag)

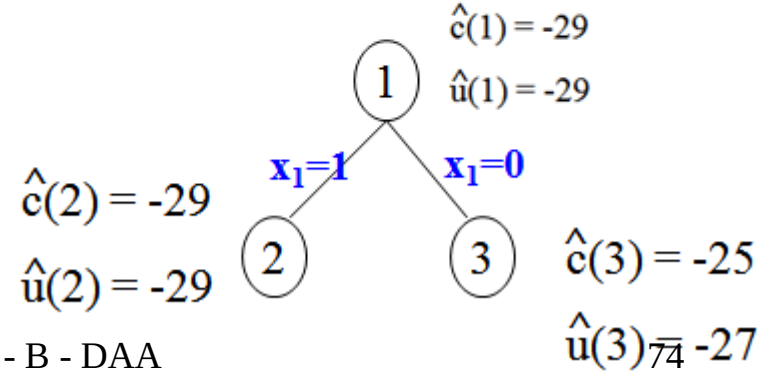
$\hat{u}(2)$ = Lower bound (2) = -10 – 15 – (2/3)*6 = -29

$\hat{c}(2)$ = Upper bound (2) = -10 – 15 – 4 = -29

For node 3: ($x_1 = 0$ means not included first item in the bag)

$\hat{u}(3)$ = Lower bound(3) = -15 – 6 – (3/4)*8 = -27

$\hat{c}(3)$ = Upper Bound(3) = – 15 – 6 – 4 = -25



Select the minimum lower bound

That is, min{ $\hat{u}(2), \hat{u}(3)$} = min {-29,-27} = -29 = $\hat{u}(2)$

There fore, choose the node 2

There fore, first item is selected, i.e $x_1 = 1$

Consider the 2nd variable to take the decision at 2nd level.

For node 4: ($x_2 = 1$ means we should place 2nd item in the bag)

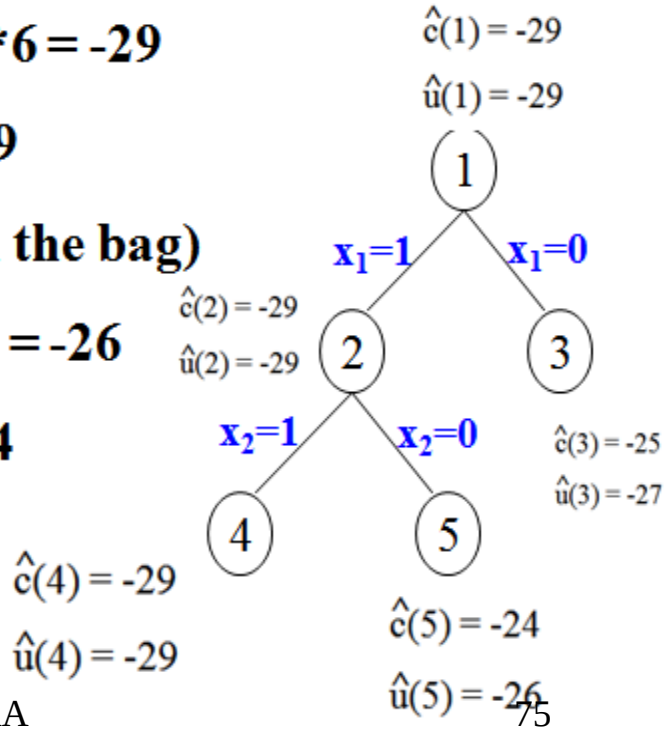
$\hat{u}(4)$ = Lower Bound(4) = -10 – 15 – (2/3)*6 = -29

$\hat{c}(4)$ = Upper bound(4) = -10 – 15 – 4 = -29

For node 5: ($x_2 = 0$ means not included 2nd item in the bag)

$\hat{u}(5)$ = Lower Bound(4) = –10–6–8–(1/2)*4 = -26

$\hat{c}(5)$ = Upper bound(4) = – 10 – 6 – 8 = -24



Select the minimum lower bound

That is, min{ $\hat{u}(4), \hat{u}(5)$} = min {-29,-26} = -29 = $\hat{u}(4)$

There fore, choose the node 4

There fore, 2nd item is selected, i.e $x_2 = 1$

Consider the 3rd variable to take the decision at 3rd level.

For node 6: ($x_3 = 1$ means we should place 3rd item in the bag)

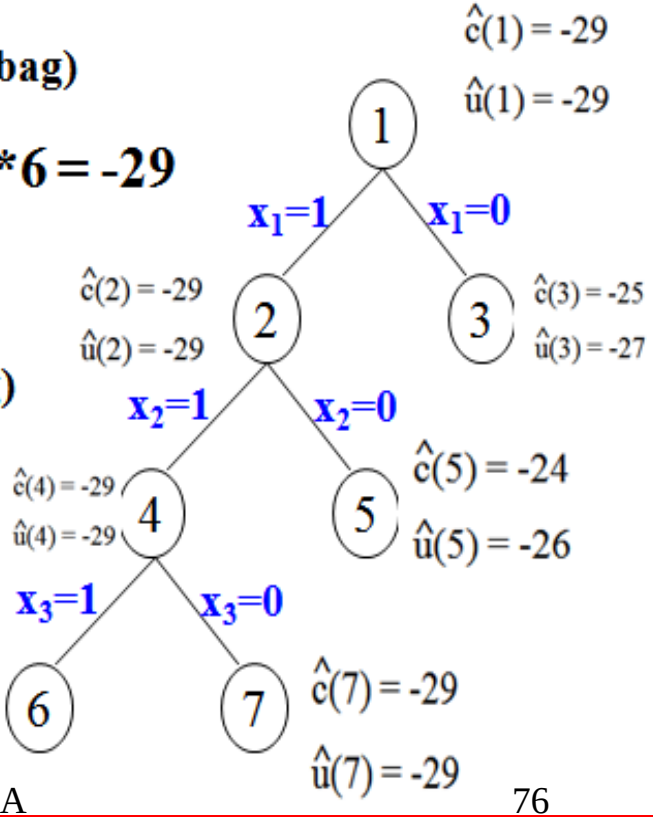
$$\hat{u}(6) = \text{Lower bound}(6) = -10 - 16 - (2/3)*6 = -29$$

$$\hat{c}(6) = \text{Upper Bound}(6) = -10 - 15 = -25$$

For node 7: ($x_3 = 0$ means not included 3rd item in the bag)

$$\hat{u}(7) = \text{Lower bound}(7) = -10 - 15 - (2/3)*6 = -29$$

$$\hat{c}(7) = \text{Upper Bound}(7) = -10 - 15 - 4 = -29$$



Select the minimum lower bound

That is, min{ $\hat{u}(6), \hat{u}(7)$} = min {-29,-29} Both are Equal.

So, select the minimum upper bound

That is, min{ $\hat{c}(6), \hat{c}(7)$} = min {-25,-29} = $\hat{c}(7)$

There fore, choose the node 7

There fore, 3rd item is not selected, i.e $x_3 = 0$

Consider the 4th variable to take the decision at 4th level.

For node 8: ($x_4 = 1$ means we should place 4th item)

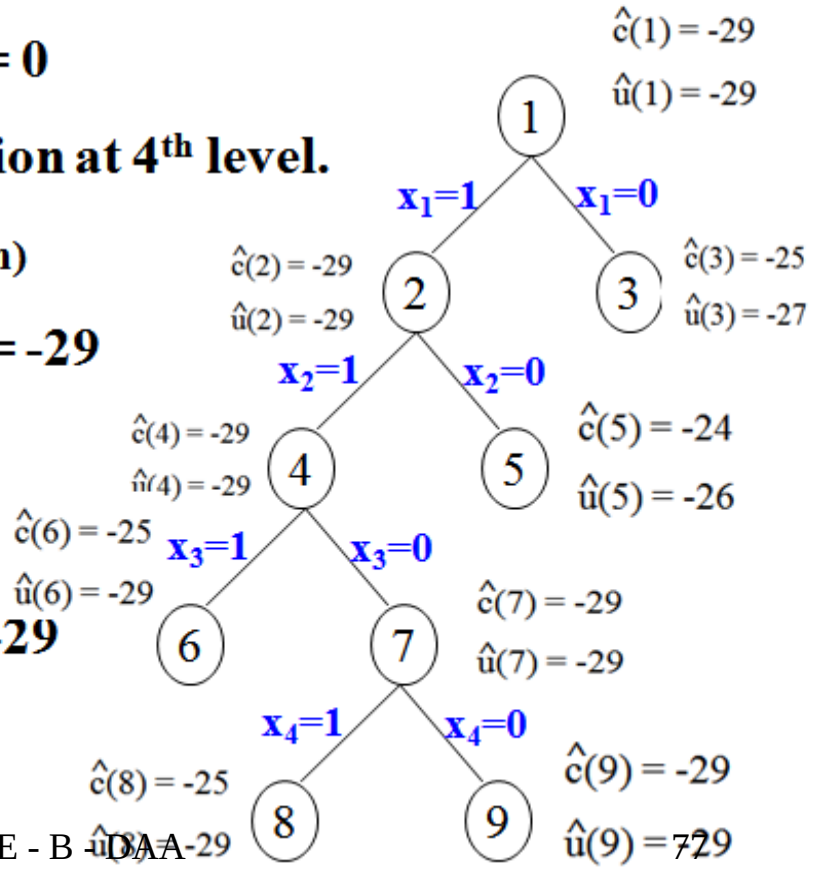
$\hat{u}(8)$ = Lower bound(8) = -10–15–(2/4)*8 = -29

$\hat{c}(8)$ = Upper bound(8)= -10–15 = -25

For node 9: ($x_4 = 0$ means not included 4th item)

$\hat{u}(9)$=Lower bound(9)= -10–15–(2/3)*6 = -29

$\hat{c}(9)$ = Upper bound(9)– 10 – 15 – 4 = -29



Since the lower bound are equal.

So, select the minimum upper bound

That is, min{ $\hat{c}(8)$, $\hat{c}(9)$} = min {-25,-29} = $\hat{c}(9)$. There fore, choose the node 9.

There fore, 4th item is not selected, i.e $x_4 = 0$

Consider the 5th variable to take the decision at 5th level.

For node 10: ($x_5 = 1$ means we should place 5th item)

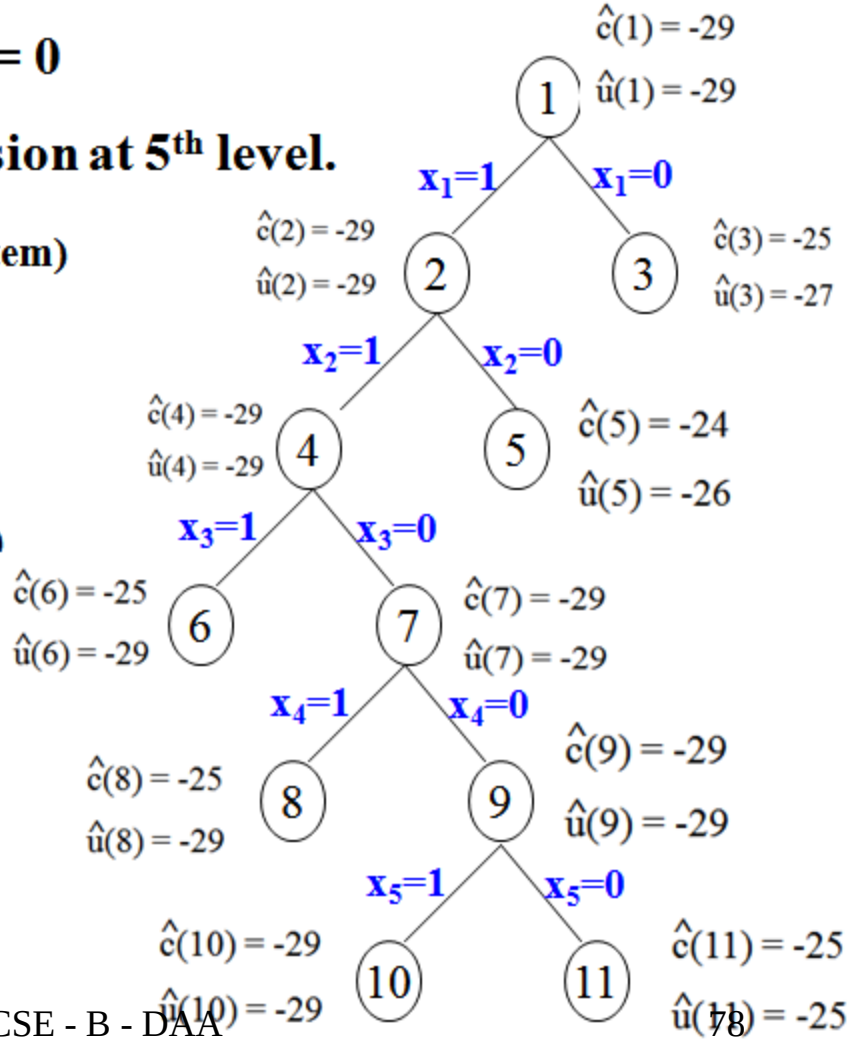
$$\hat{u}(10) = -10 - 15 - 4 = -29$$

$$\hat{c}(10) = -10 - 15 - 4 = -29$$

For node 11: ($x_5 = 0$ means not included 5th item)

$$\hat{u}(11) = -10 - 15 = -25$$

$$\hat{c}(11) = -10 - 15 = -25$$



So, select the minimum lower bound

That is, min{ $\hat{u}(10)$, $\hat{u}(11)$} = min {-29,-25} = -29 = $\hat{u}(10)$.

There fore, choose the node 10. There fore, 5^{th} item is selected, i.e $x_5 = 1$

There fore the path is 1 – 2 – 4 - 7 – 9 – 10

The solution for Knapsack is $(x_1, x_2, x_3, x_4, x_5)$=(1, 1, 0, 0, 1)

Maximum Profit = 10 + 15 + 4 = 29

****

Steps to be followed in LCBB

Step 1:Compute $\hat{c}(.)$ and u(.) for a generated node, then if $\hat{c}$>upper(initially upper=u(root node)) immediately kill the node otherwise add the node to the list of live nodes.

Step 2: Update upper = u(.), if upper is greater than u(.).

Step 3: The Least Cost node will be the next E-node.

Step 4: Whenever a node about become an E-node, check if $\hat{c}(.)$ is greater than upper, if yes kill the node otherwise generate its children. Also kill the nodes which are not feasible.

Step 5: Continue step 1 to 4 till all the live nodes are expanded.

Step 6 : Finally the highest U value (absolute value) node will be the answer node. Trace the path in backward direction from answer node to root node for solution subset.

Example:- Draw the portion of state space tree generated by LCBB by the following Knapsack Problem. N=4, $(p_1,p_2,p_3,p_4)=(10,10,12,18)$ $(w_1,w_2,w_3,w_4)=(2,4,6,9)$ and m=15

Solution: First convert the profits to negative. $(p_1,p_2,p_3,p_4)=(-10,-10,-12,-18)$

The total profits must be maximization. So, it is maximization problem.

Calculate the lower bound and upper bound for each node. Lower bound and upper bound are calculated as sum of $P_i.X_i$. The sum is always <=M.

But in lower bound calculation the fractions are allowed. In upper bond calculations fractions are not allowed.

May be solution in the form of S ={x1,x3} or S= {1,0,1,0}

Place the first item in the bag. That is $w_1 = 2$

Remaining weight is 15 – 2 = 13.

Now, Place the 2nd item in the bag. That is $w_2 = 4$

Remaining weight is 13 – 4 = 09.

Now, Place the 3rd item in the bag. That is $w_3 = 6$

Remaining weight is 09 – 6 = 03.

Since fractions are not allowed in calculation of upper bound, so we can not place the third and 4th item.

Initially, the upper bound is ∞.

There fore profits earned = -10 -10-12 = -32 = Upper bound

Therefore, ∞ > -32. the upper bound is -32

To calculate the lower bound, place the third item in the bag. Since fractions are allowed.

(1) $\hat{u}(1) = -32$, $\hat{c}(1) = -38$

Lower bound = -10 -10-12-(3/9) X 18 = -38

For Node 2: (x_1 item included. That is x_1=1)

Lower bound (2) = $\hat{c}(2)$ = -10 -10-12-(3/9) X 18 = -38

Upper bound(2) = $\hat{u}(2)$ = -10 -10-12 = -32

For Node 3: (x_1 item not included. That is x_1=0)

Lower bound (3) = $\hat{c}(3)$ = -10-12-(5/9) X 18 = -32

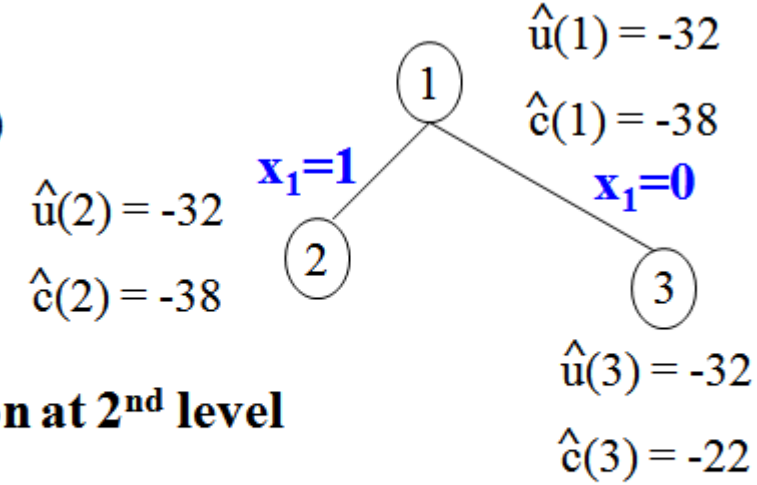
Upper bound(3) = $\hat{u}(3)$ = -10-12 = -22

Select the minimum cost. That is

Min{$\hat{c}(2), \hat{c}(3)$} = {-38, -32} = -38, that is c(2)

Therefore choose the node 2.

Therefore, select the first item, i.e x_1=1



Consider the 2nd variable to take the decision at 2nd level

For Node 4: (x_2 item included. That is x_2=1)

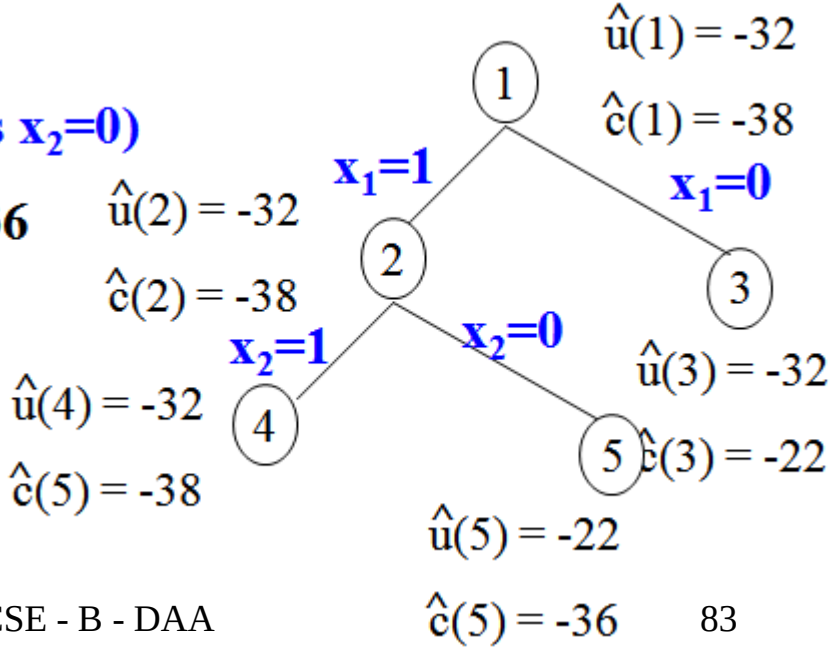
Lower bound (4) = $\hat{c}(4)$ = -10 -10-12-(3/9) X 18 = -38

Upper bound(4) = $\hat{u}(4)$ = -10 -10-12 = -32

For Node 5: (x_2 item not included. That is x_2=0)

Lower bound(5)=$\hat{c}(5)$=-10-12-(7/9)X18=-36

Upper bound(5) = $\hat{u}(5)$ = -10-12 = -22

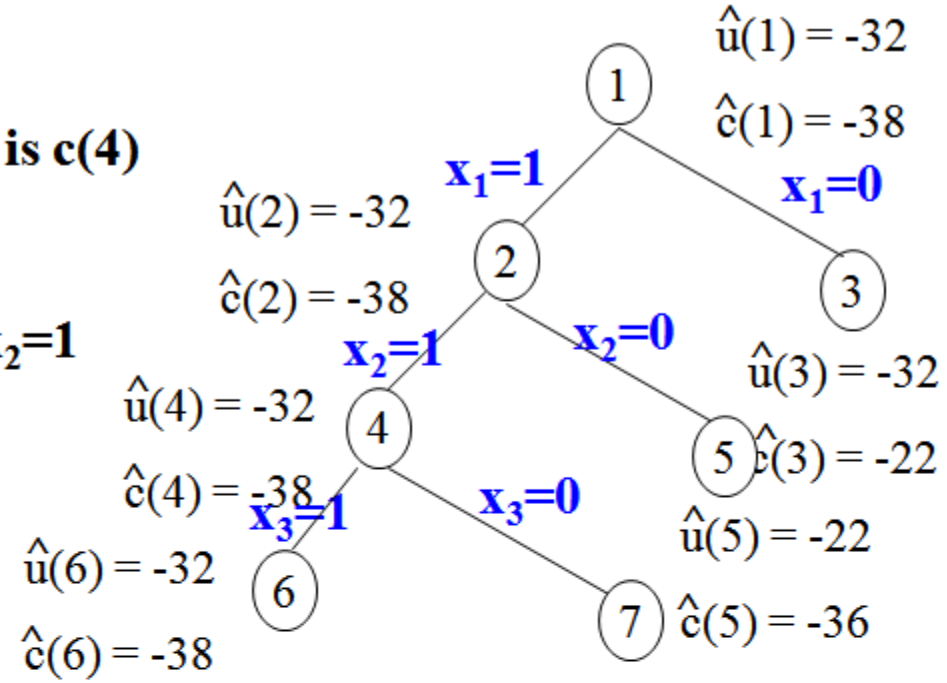


Select the minimum cost. That is

Min{$\hat{c}(4), \hat{c}(5)$} = {-38, -36} = -38, that is c(4)

Therefore choose the node 4.

Therefore, select the second item, i.e x_2=1



Consider the 3rd variable to take the decision at 3rd level

For Node 6: (x_3 item included. That is x_3=1)

Lower bound (6) = $\hat{c}(6)$ = -10 -10-12-(3/9) X 18 = -38

Upper bound(6) = $\hat{u}(6)$ = -10 -10-12 = -32

For Node 7: (x_3 item not included. That is x_3=0)

Lower bound(7)=$\hat{c}(7)$=-10-10-18=-38

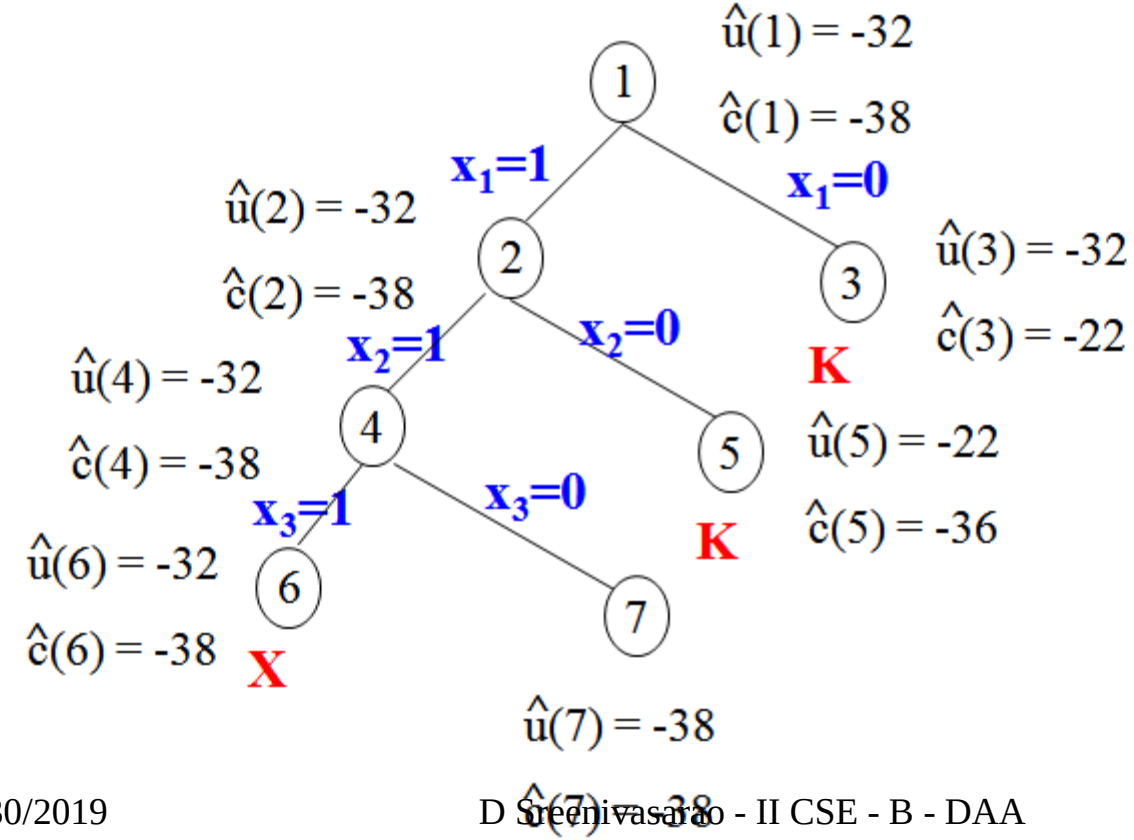
Upper bound(7) = $\hat{u}(7)$ = -10-10-18 = -38

Therefore, expand the node 7.

The cost of node 3 is -22. -22 > - 38. so, node 3 is killed.

Similarly, The cost of node 5 is -36. -36 > - 38. so, node 5 is killed.

If we include first, second, third and fourth items. The weight is 2 + 4 + 6 + 9 = 21. so, 21 > 15. therefore, Node 6 not produce the solution. It is infeasible.



Therefore choose the node 7 and expand it.

Therefore, 3rd item is not selected i.e $x_3=0$

Consider the 4th variable to take the decision at 4th level

For Node 8: (x_4 item included. That is $x_4=1$)

Lower bound(8)=c(8)=-10-10-18= - 38

Upper bound(8) = u(8) = -10-10-18 = - 38

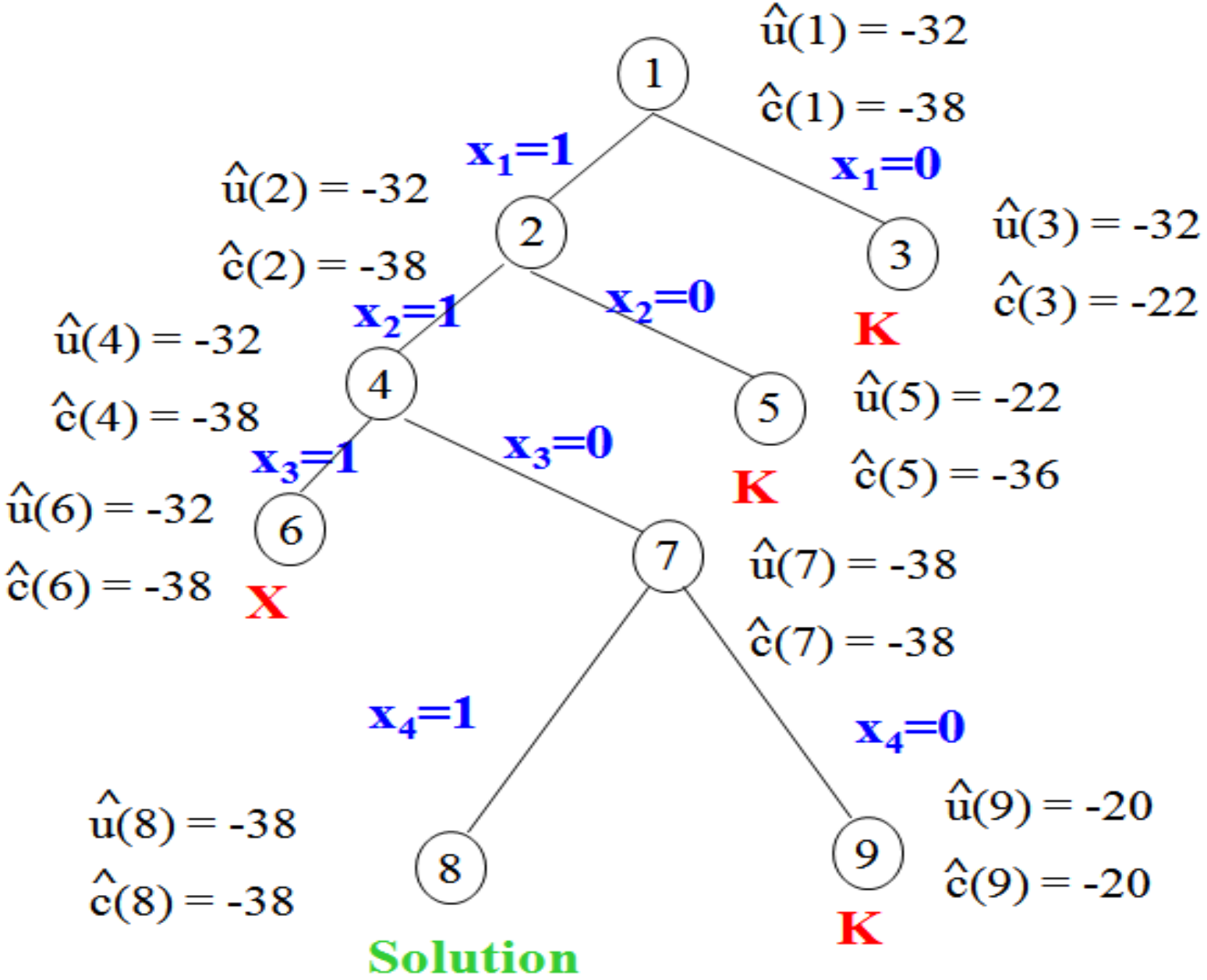
For Node 9: (x_4 item not included. That is $x_4=0$)

Lower bound(9)=c(9)=-10-10 =-20

Upper bound(9) = u(9) = -10-10 = - 20

The cost of node 9 is -20. -20 > - 38. so, node 9 is killed.

The node 8 is alive node.


1
$\hat{u}(1) = -32$
$\hat{c}(1) = -38$
$x_1=1$
$x_1=0$
2
$\hat{u}(2) = -32$
$\hat{c}(2) = -38$
3
$\hat{u}(3) = -32$
$\hat{c}(3) = -22$
K
$x_2=1$
$x_2=0$
4
$\hat{u}(4) = -32$
$\hat{c}(4) = -38$
5
$\hat{u}(5) = -22$
$\hat{c}(5) = -36$
K
$x_3=1$
$x_3=0$
6
$\hat{u}(6) = -32$
$\hat{c}(6) = -38$
X
7
$\hat{u}(7) = -38$
$\hat{c}(7) = -38$
$x_4=1$
$x_4=0$
8
$\hat{u}(8) = -38$
$\hat{c}(8) = -38$
9
$\hat{u}(9) = -20$
$\hat{c}(9) = -20$
K
Solution

The solution is

The sub set form : S = {x_1, x_2, x_4}

The included or not included form: S = x{1,1,0,4}

The profit is -10 + -10 + 0 + -18 = -38

The maximum profit is 38

And their weights are 2 + 4 + 0 + 9 = 15.

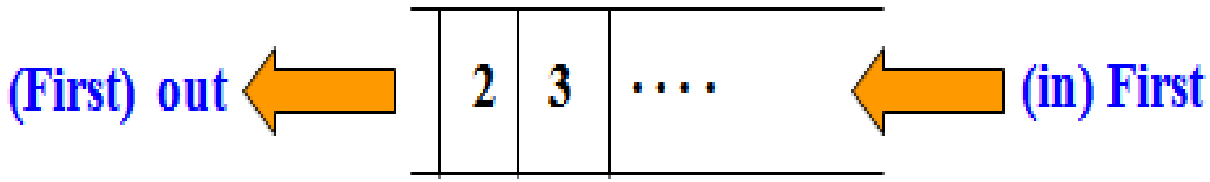
Example: consider Knapsack with n=4, $(v_1,v_2,v_3,v_4)=(10,10,12,18)$ $(w_1,w_2,w_3,w_4,w_5)=(2,4,6,9)$ and m=15

Solution : The FIFOBB algorithm proceeds with node 1 as the root node and make it as E-node.

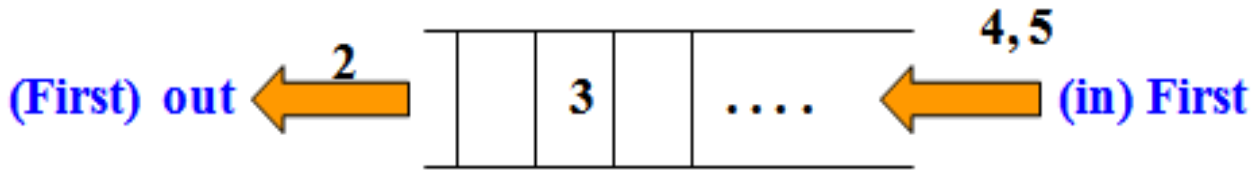
FIFOBB initializes u(1) = -10 -10 -12 = -32

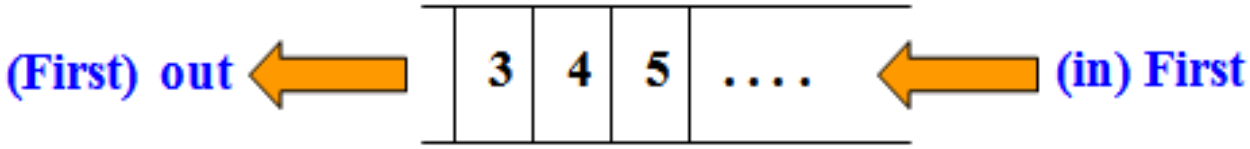
Hence nodes 2 and 3 are generated

There fore node 2 and 3 are sent to Queue.

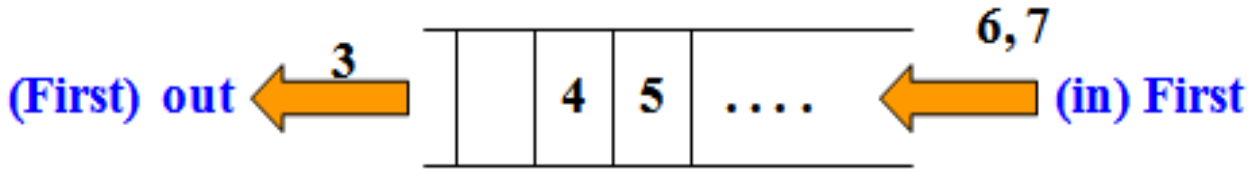


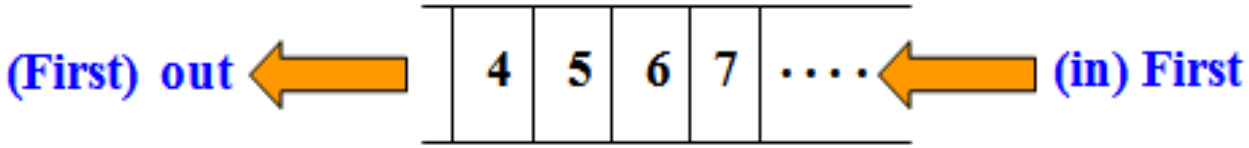
As 2 staying first in the Queue, it becomes E-node and it produces node 4 and 5 as children . Hence the Queue becomes





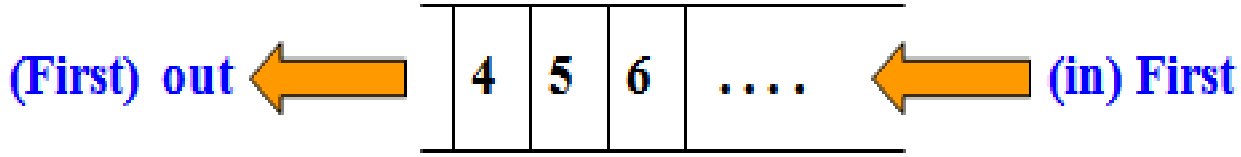
Node 3is the next E-node and it produces node 6 and 7 as children . Hence the Queue becomes

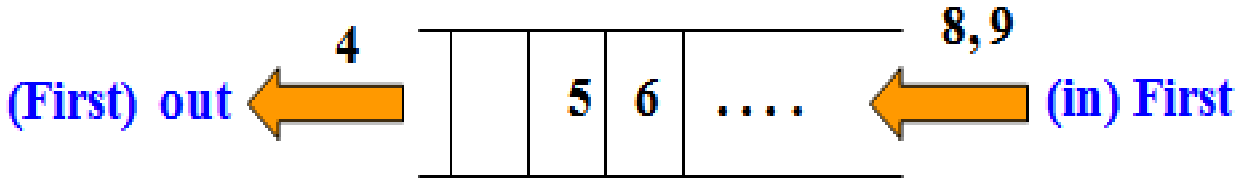


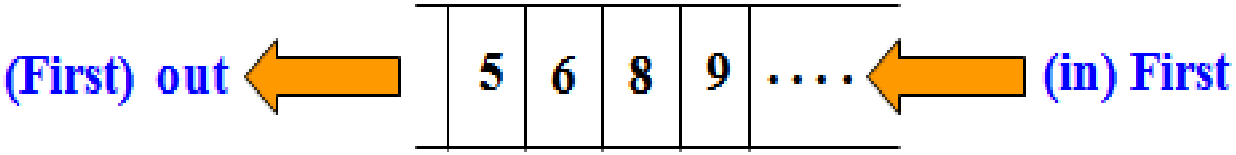


The node 7 is immediately killed because c(7)> upper bound.

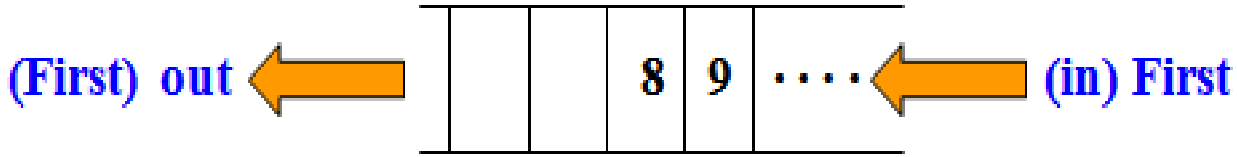
Then nodes 8 and 9 are produced from the current E-node 4. also the upper bound is updated to -38, u(9) = -38. now the Queue is





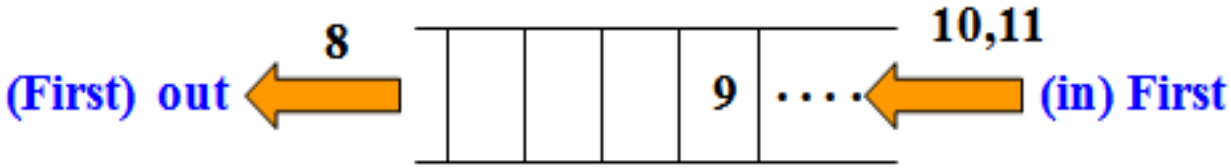


Neither node 5 nor 6 are expanded, because each c(x)> upper bound and those are killed.

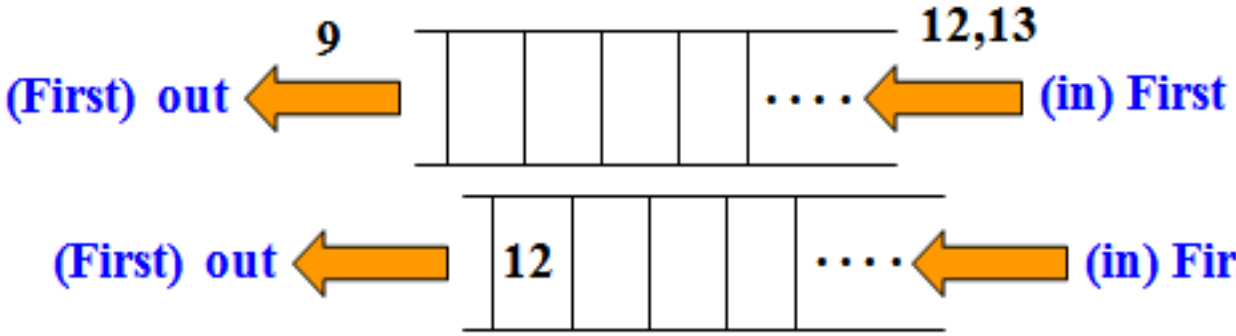


Hence node 8 is expanded and its children are 10 and 11, both are killed.

Node 10 is infeasible , so killed.



The next node from the queue is 9 and is expanded to 12 and 13, node 13 is killed. The node 12 is inserted in to the Queue.



The only live node in Queue is 12 and search terminates.

Home Work :-

1.Draw the portion of the state space tree generated by LCBB and FIFIBB for the following knapsack problem.

a)n=5, (p1,p2,p3,p4,p5)=(10,15,6,8,4), (w1,w2,w3,w4,w5)=(4,6,3,4,2) and m=12

b) n=5, (p1,p2,p3,p4,p5)=(w1,w2,w3,w4,w5)=(4,4,5,8,9) and m=15.