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

5.2.5. Maze generation Problem

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 $(x_{i+1}, \ldots.x_n)$ solution may be ignored completely.

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.

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.

Basic Terminology:

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

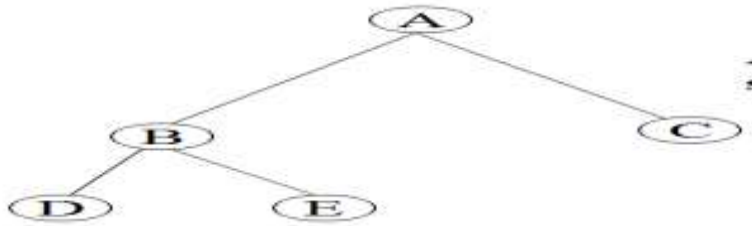
Example: in Knapsack problem the explicit constraints are $x_i = 0$ or 1 (or) $0 \leq x_i \leq 1$

2. 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.

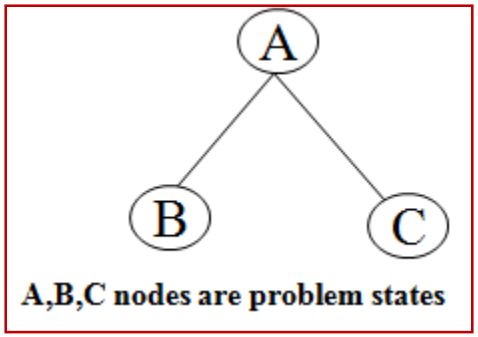
3. Criterion Function: it is a function $P(x_1,x_2,x_3,\ldots,x_n)$ which needs to be maximized or minimized for a given function.

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

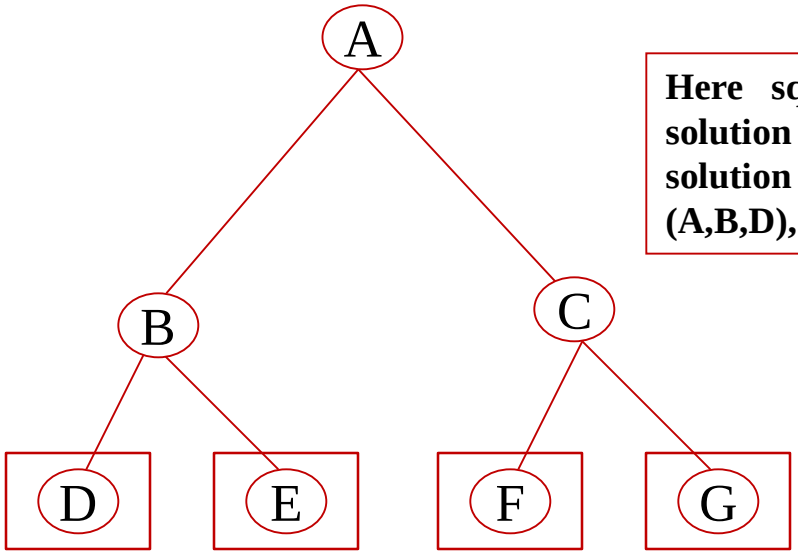


ABD, ABE, AC are the tuples in solution Space

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

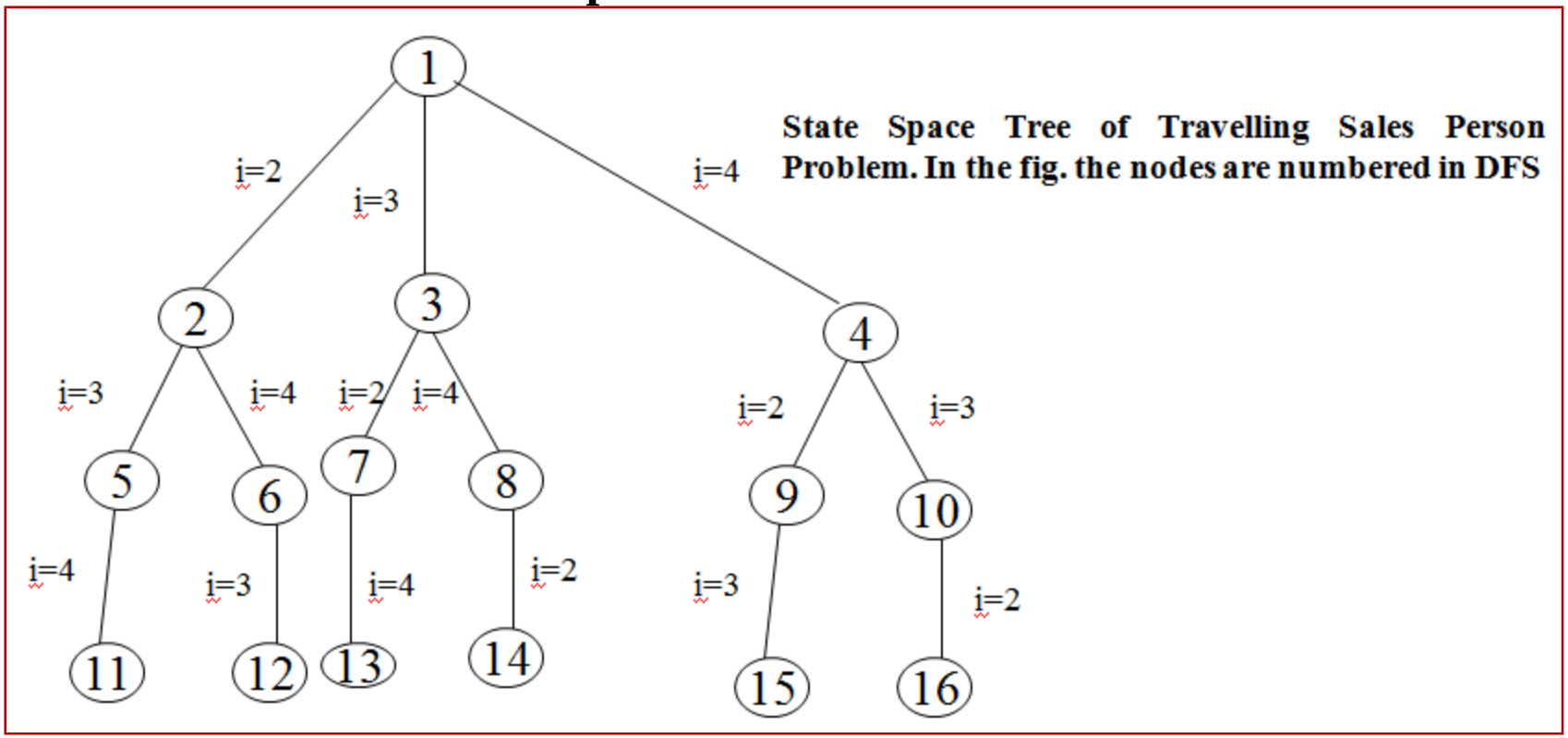


6. Solution States: **these are those problem states S for which the path from the root to S define a tuple in the solution space.**



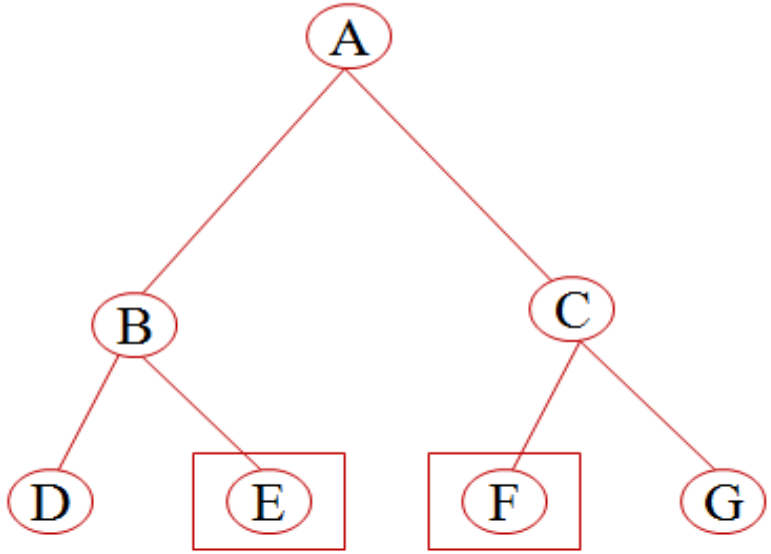
Here square nodes indicates solution. For the solution space, there exist 4 solution states. These solution states represented in the form of tuples (A,B,D), (A,B,E), (A,C,F) and (A,C,G)

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



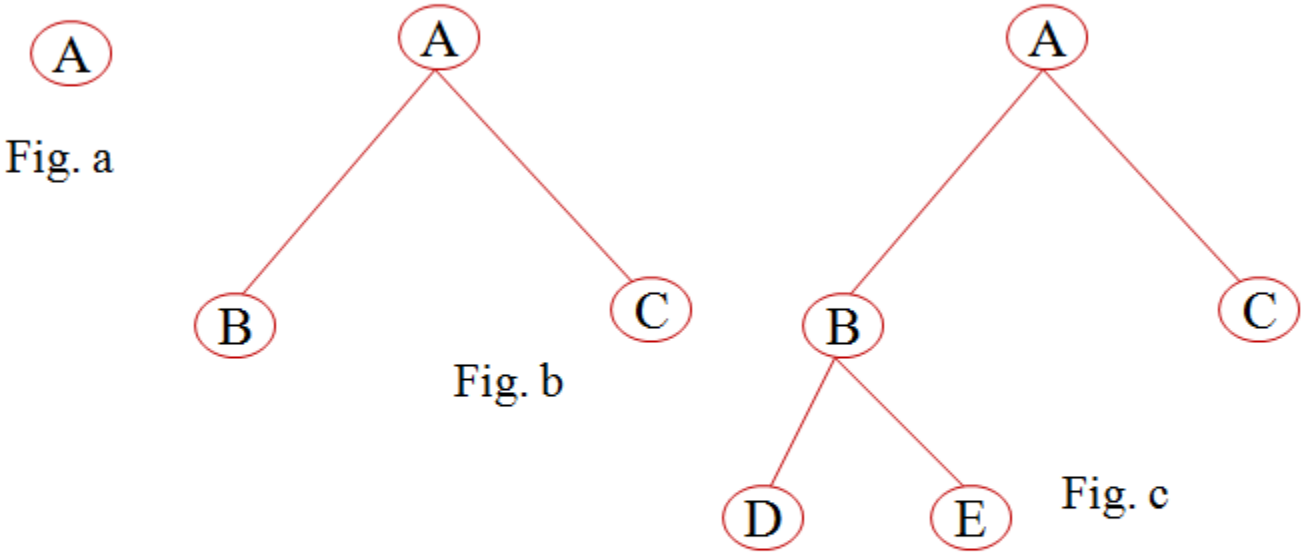
State Space Tree of Travelling Sales Person Problem. In the fig. the nodes are numbered in DFS

8. 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

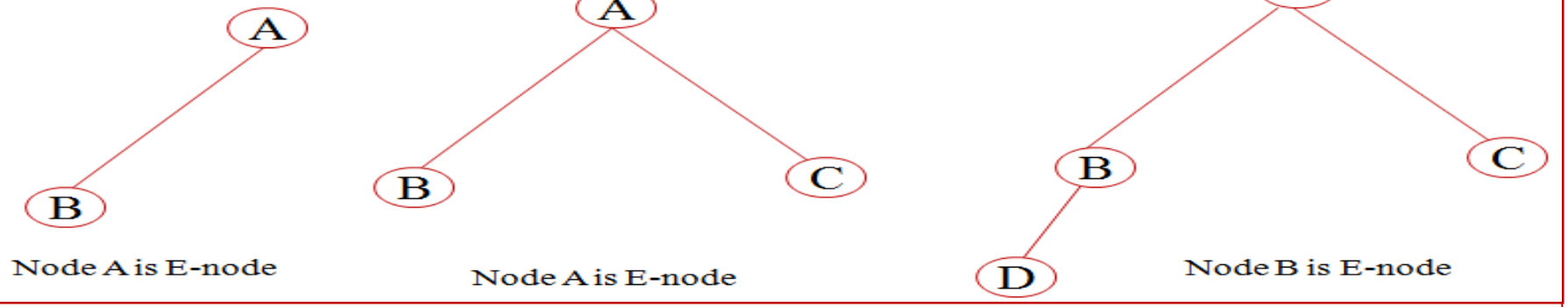


In fig.(a) node A is called live node since the children of node A have not yet been generated.

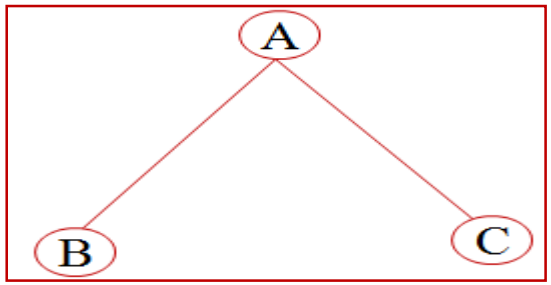
In fig.(b) node A is not a live node but node B an C are live nodes

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

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



11. 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 (Bk(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		

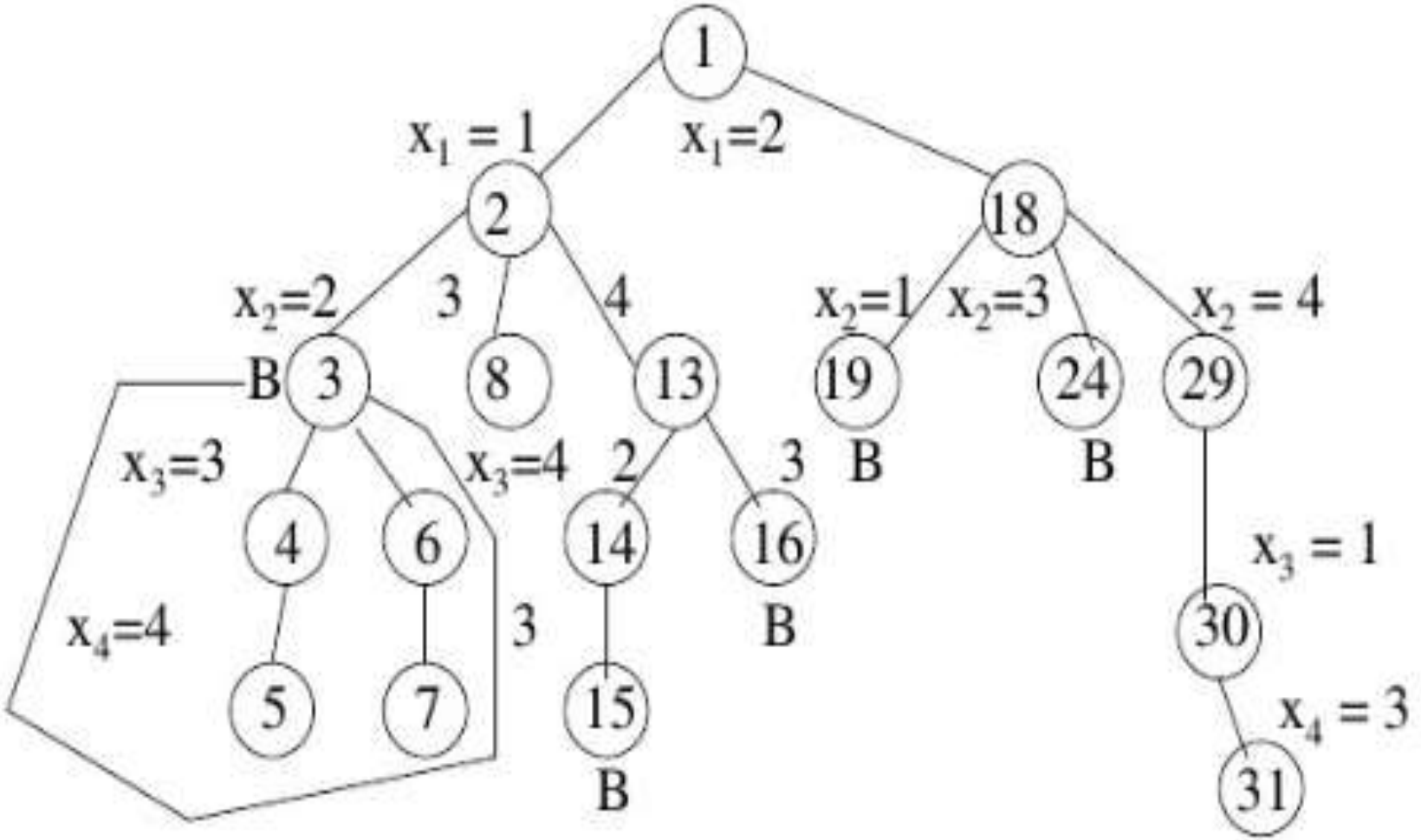
X	Q_1	X	X
X	X	X	Q_2
	X	X	X
	X		X

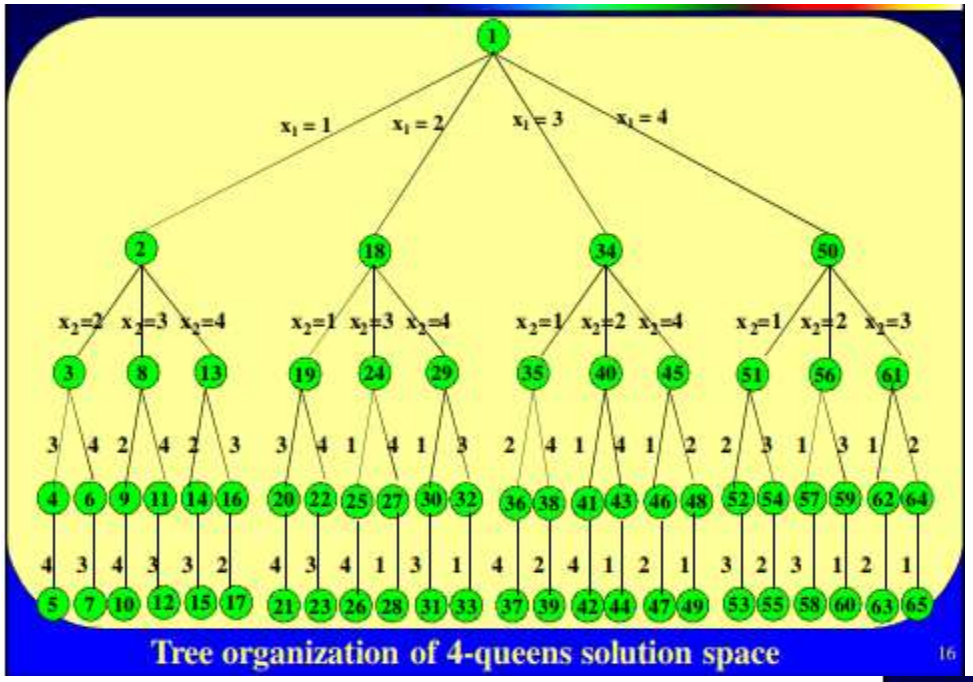
X	Q_1	X	X
X	X	X	Q_2
Q_3	X	X	X
X	X		X

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)

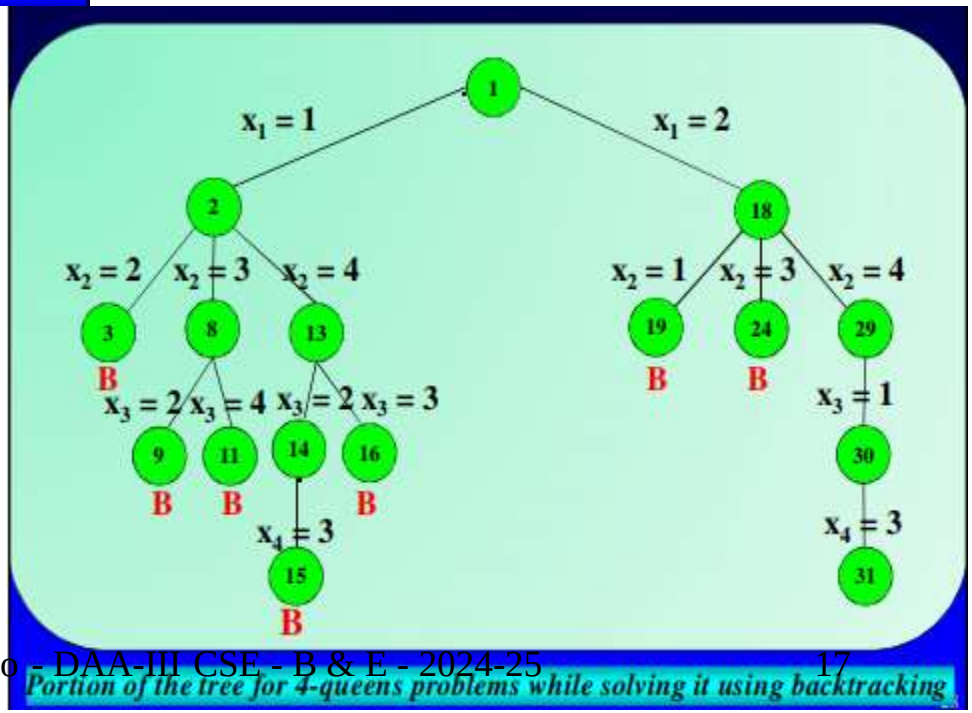
State space tree: 4 Queens problem





Tree organization of 4-queens solution space

16



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.

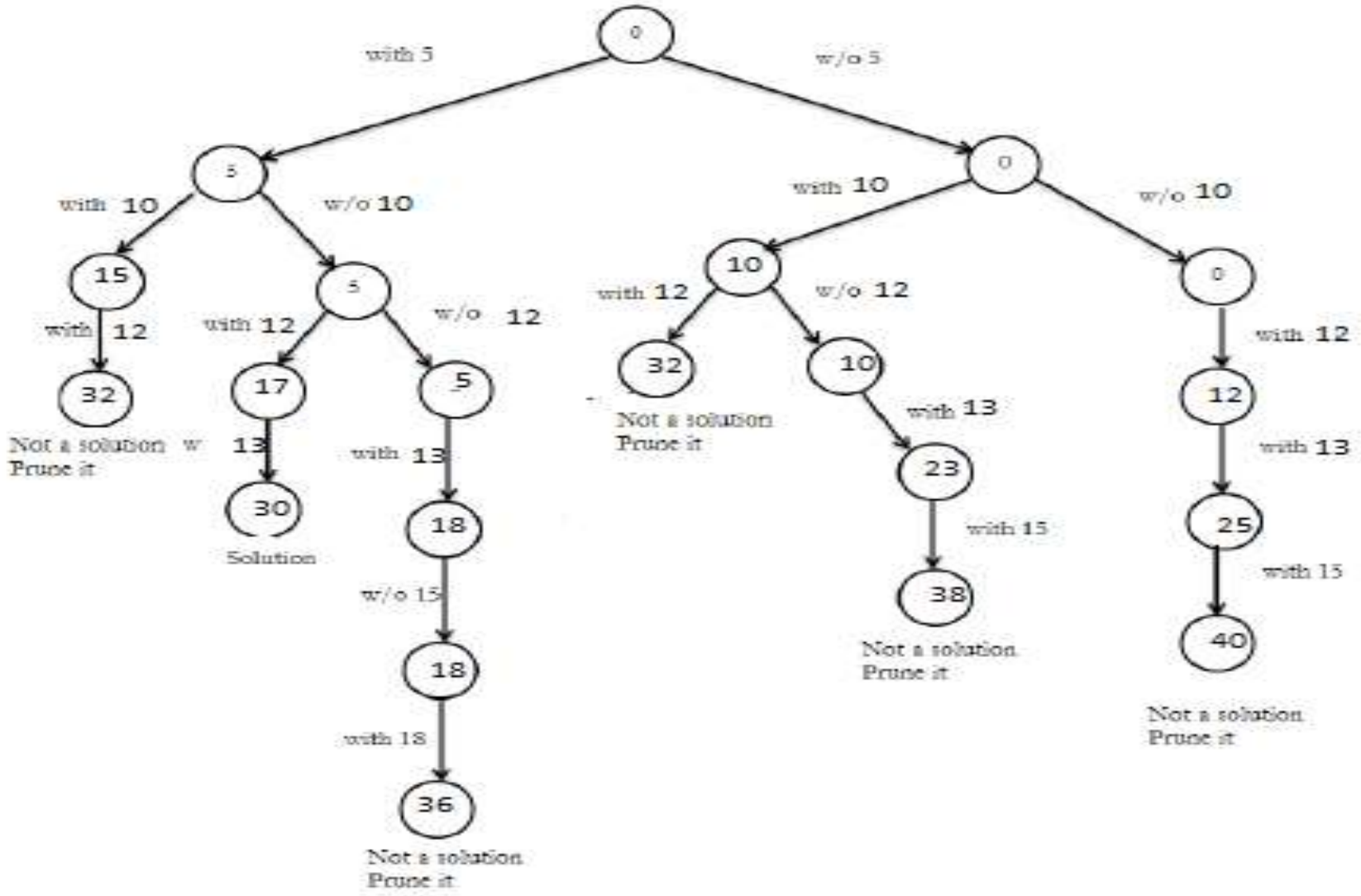
Step 5: if the sub set is feasible then repeat step 2.

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

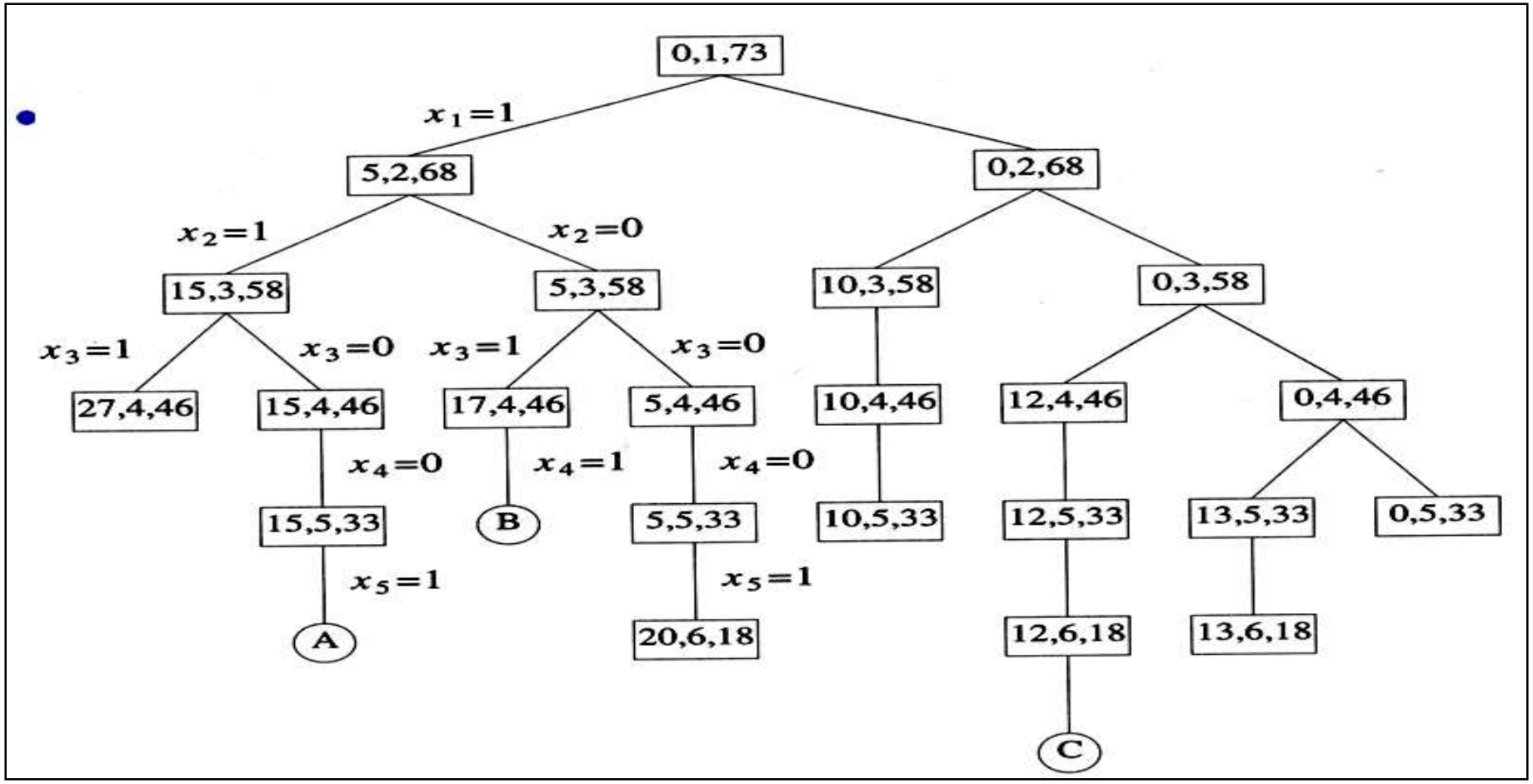
Example: consider a set S={5,10,12,13,15,18} and d=30. solve it for obtaining sum of subsets.

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

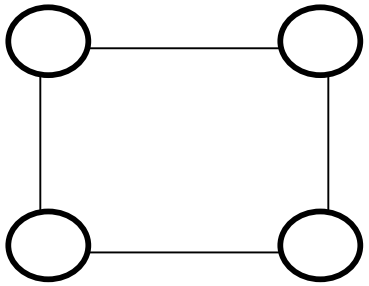
Let G is a graph and m be the given positive integer.

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".

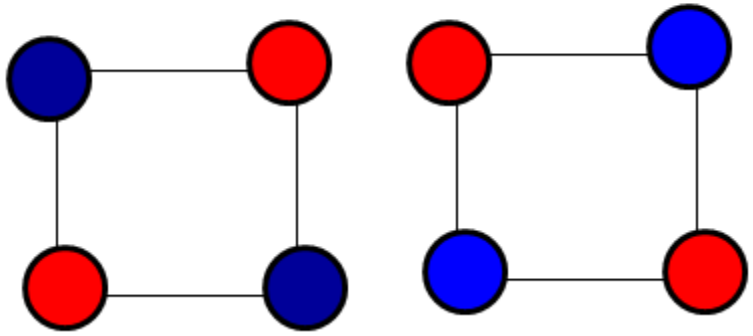
Note: If d is the degree of the given graph, then it can be colored with d+1 colors.

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.

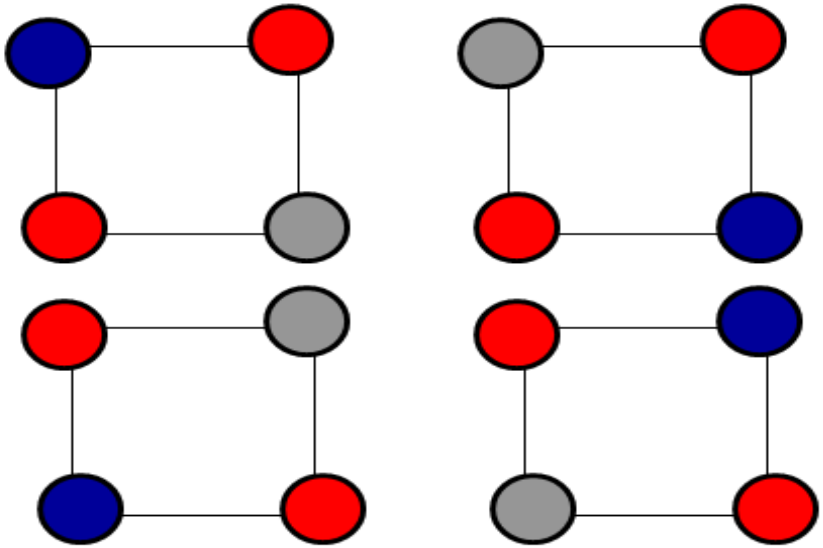
Example:



Number of Possible ways : 2



No.of Colors used: 2

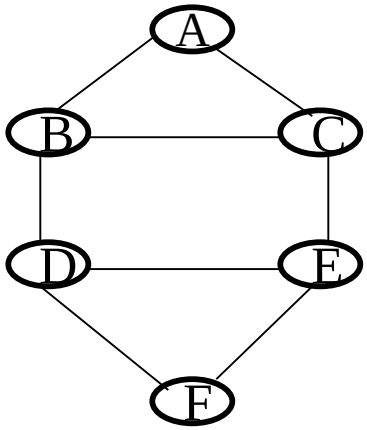


Some possible ways

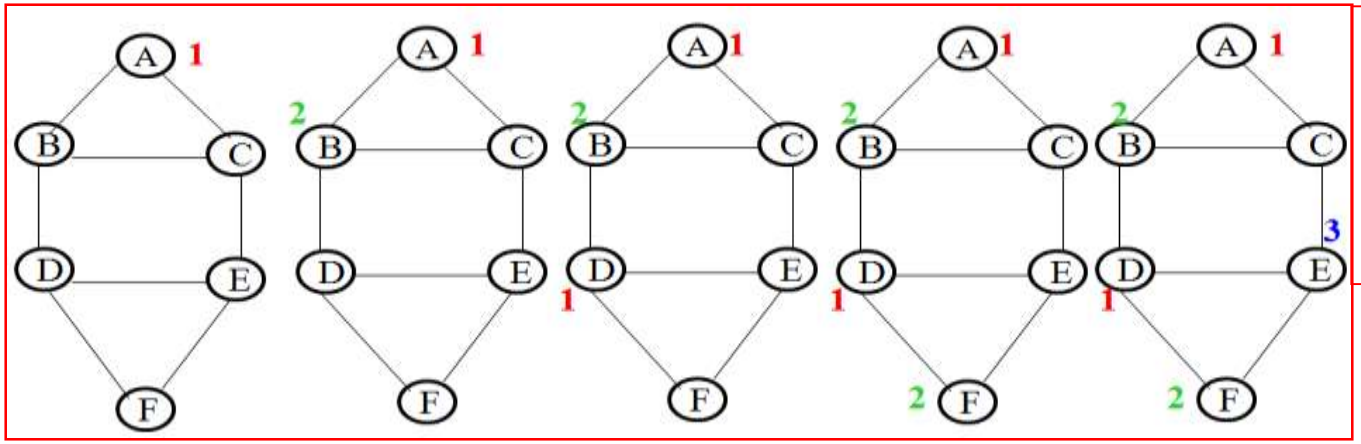
No.of Colors used: 3

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

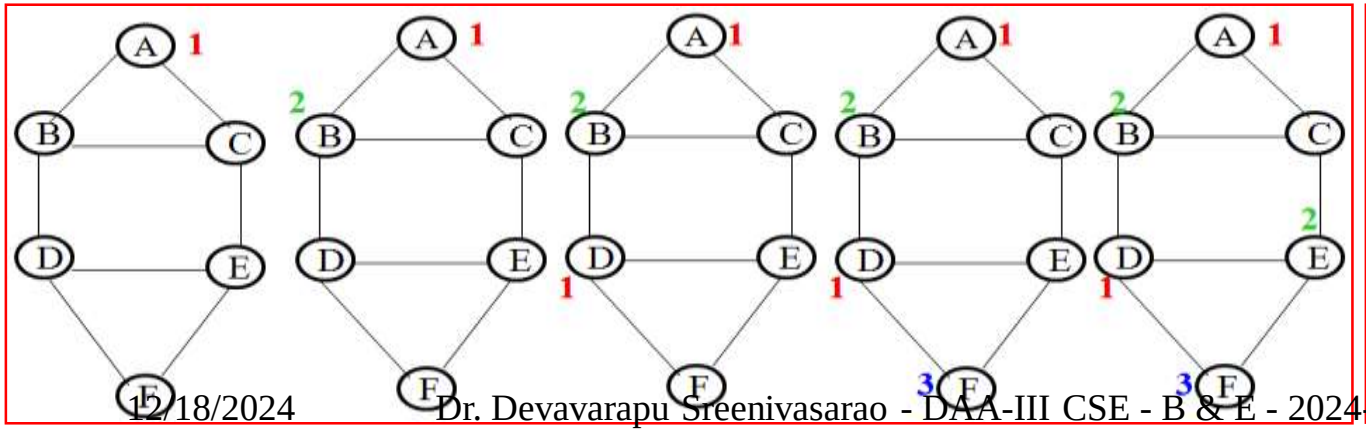
Example:

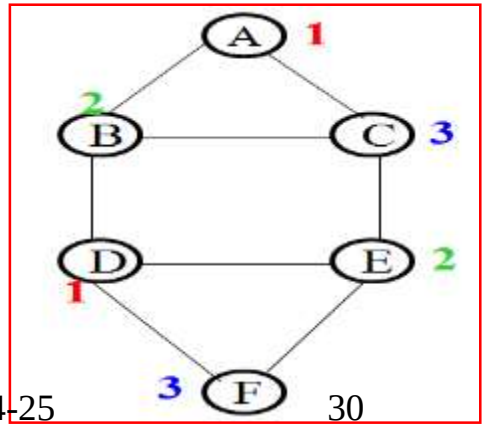


Step : A Graph G consists of vertices from A to F. there are three colors used RED, GREEN, BLUE. We will number them out. That means 1 indicates RED, 2 indicates GREEN and 3 indicates BLUE color.

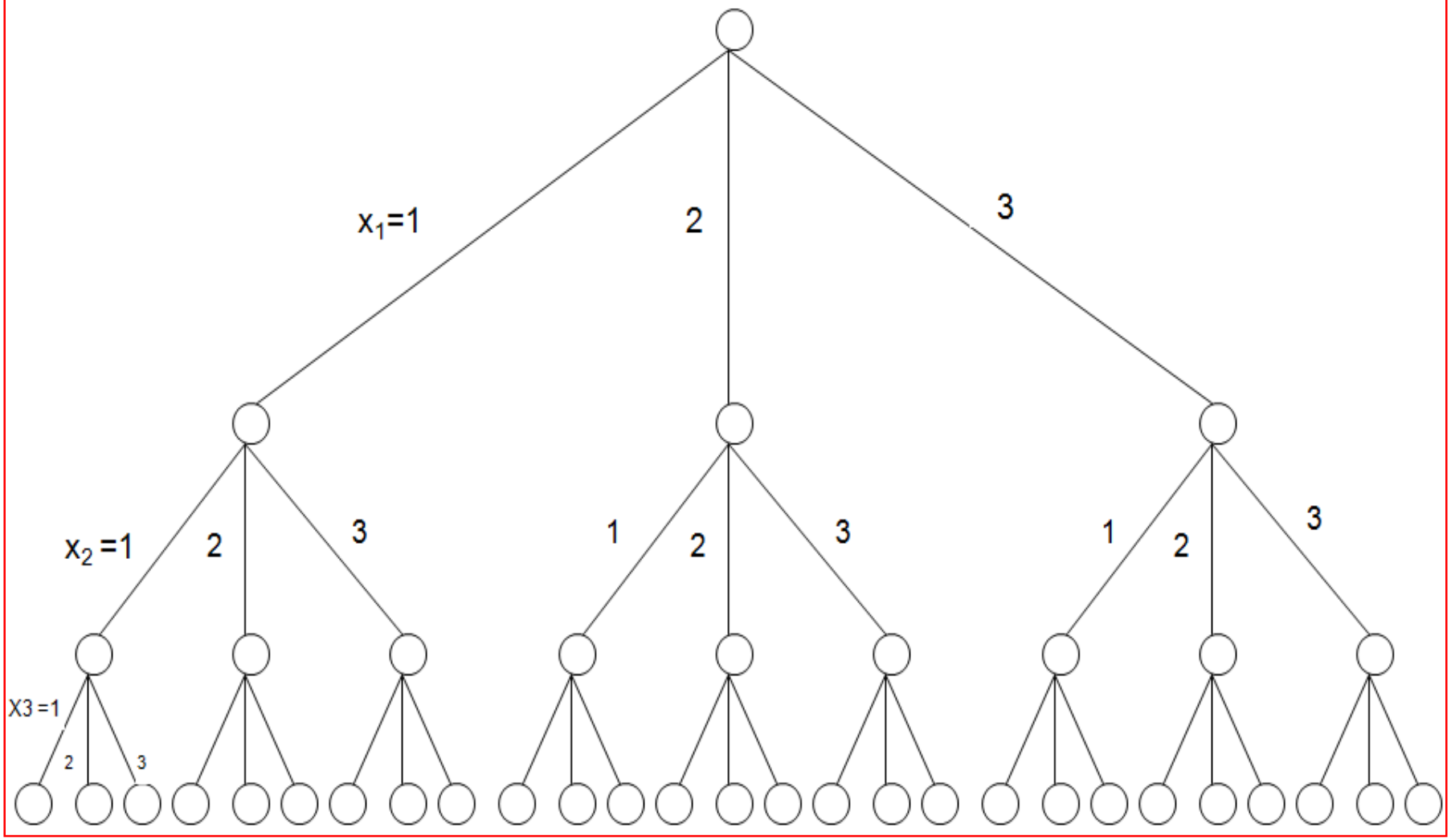


Stuck here!!
Can not assign 1 or 2 or 3. hence back track

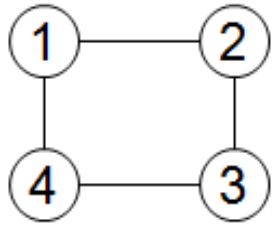


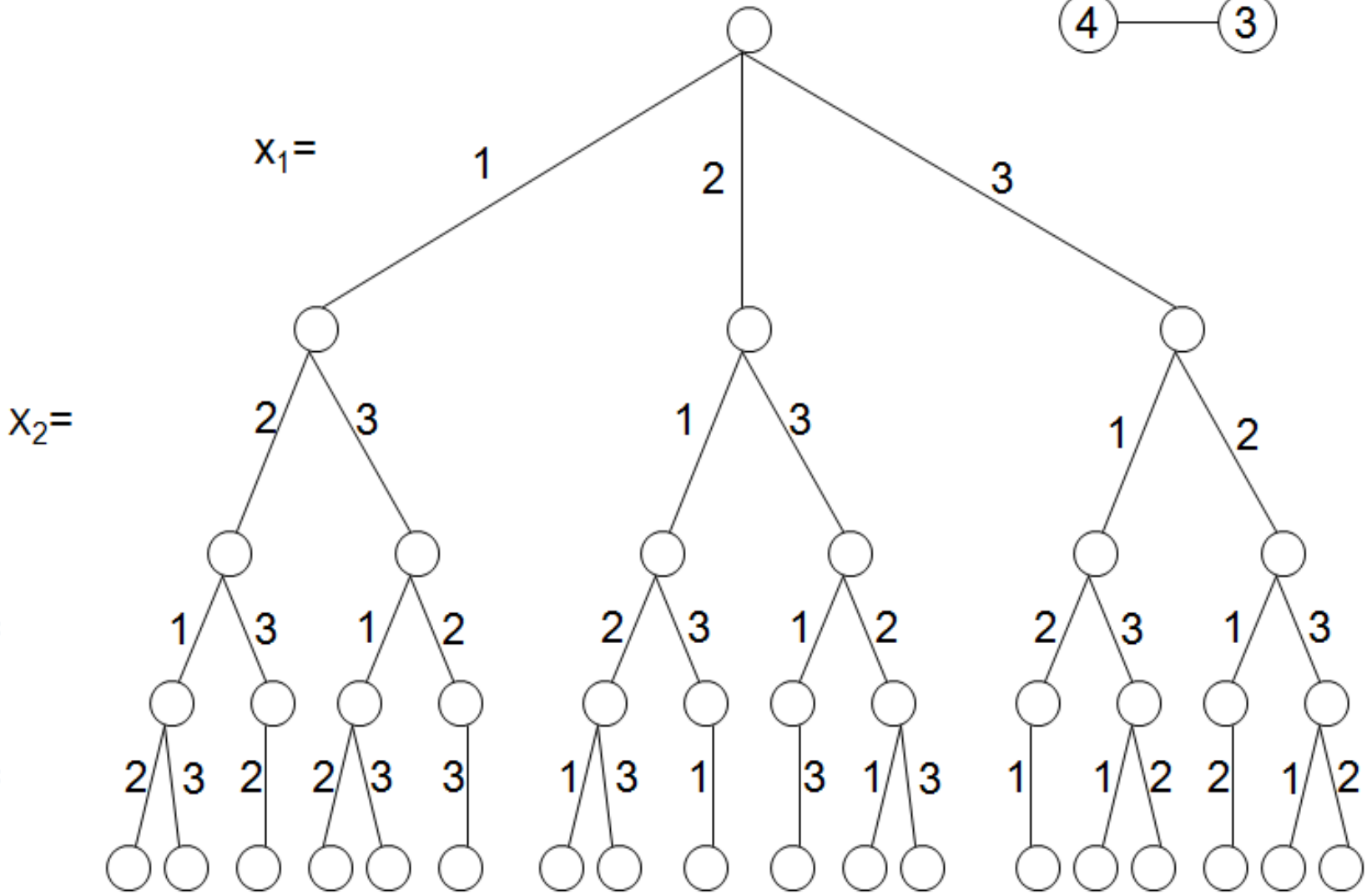


Solution space tree for mColoring when n=3 and m=3



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

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

$$= n.\, m^n$$

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

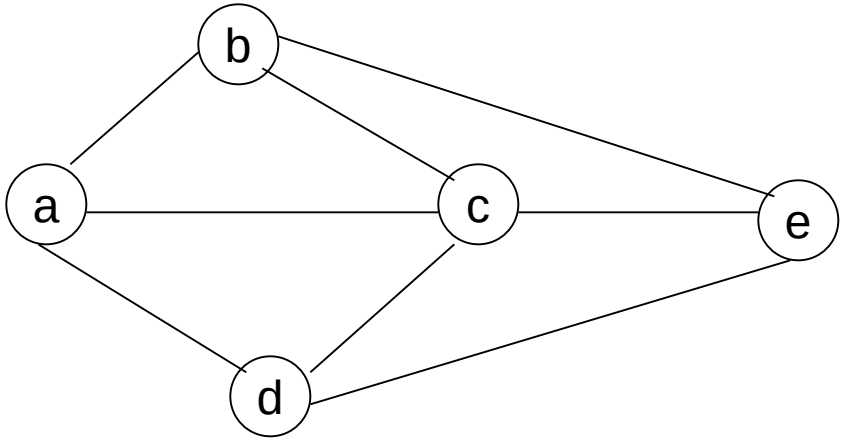
5.2.4.Hamiltonian Cycle:

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.

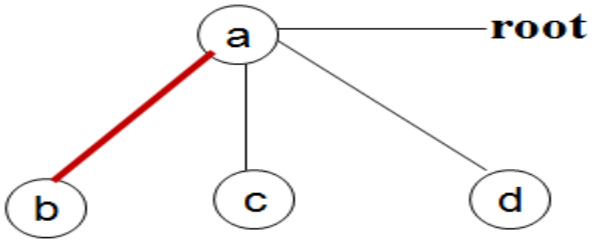
A circuit $x_0,x_1,\ldots.x_{n-1},x_n$, x0(with n>1) in a graph G=(V,E) is called Hamiltonian circuit if x_0, $x_1,\ldots.x_n$ is a Hamiltonian Path.

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

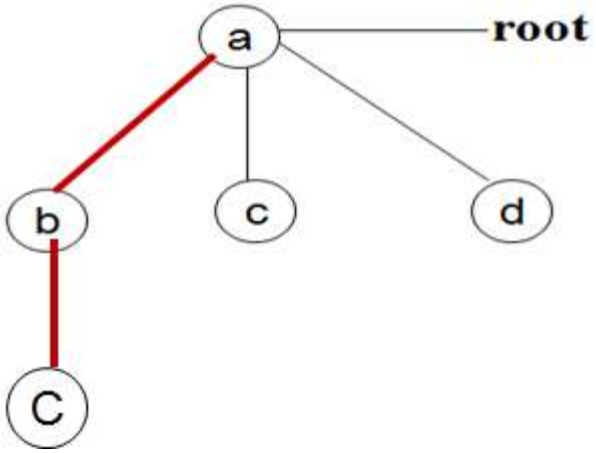


Step 1: initially we start our search with vertex ‘a’. The vertex ‘a’ becomes the of our implicit tree.

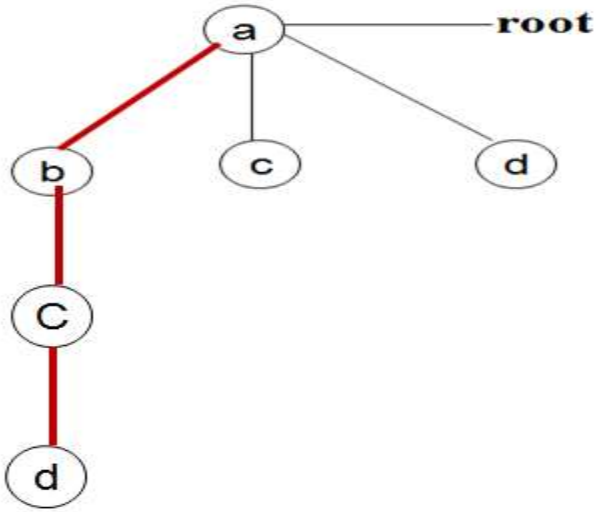
Next, we chose vertex ‘b’ adjacent to ‘a’ as it comes first in lexicographical order or (b,c,d)



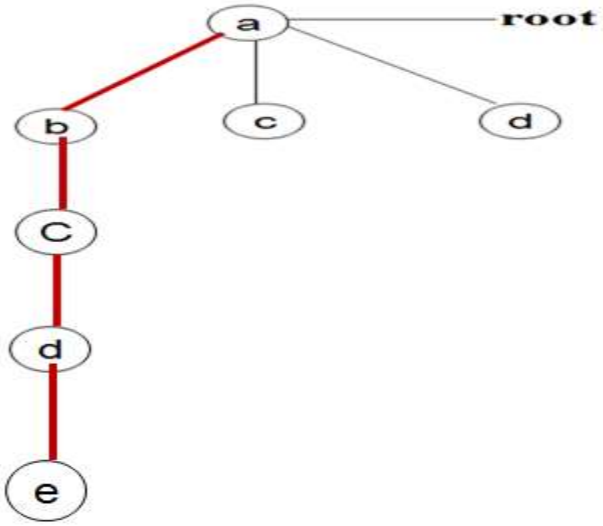
Next vertex ‘c’ is selected which is adjacent to ‘b’ and which comes first in lexicographical or (c,e)



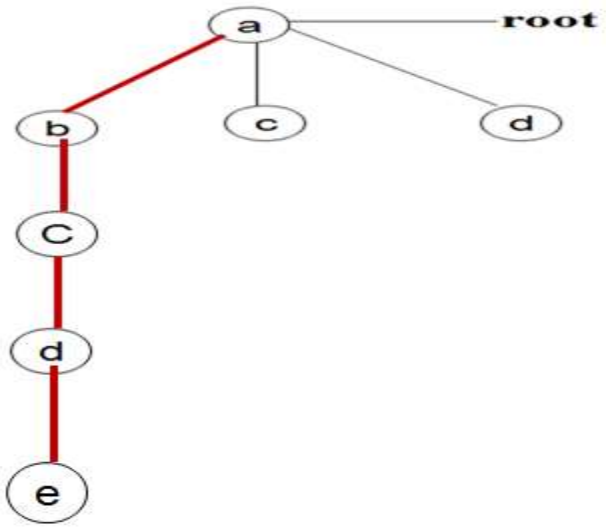
Next vertex ‘d’ is selected which is adjacent to ‘c’ and which comes first in lexicographical or (d,e)



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

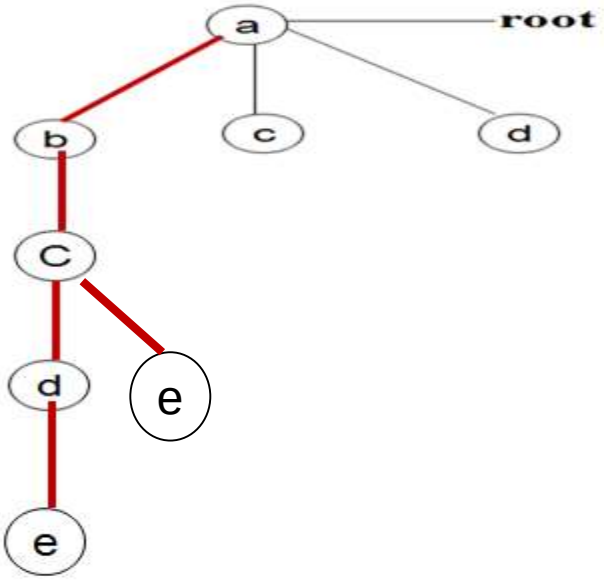
So, we back track one step and remove the vertex ‘e’ from our partial solution.

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

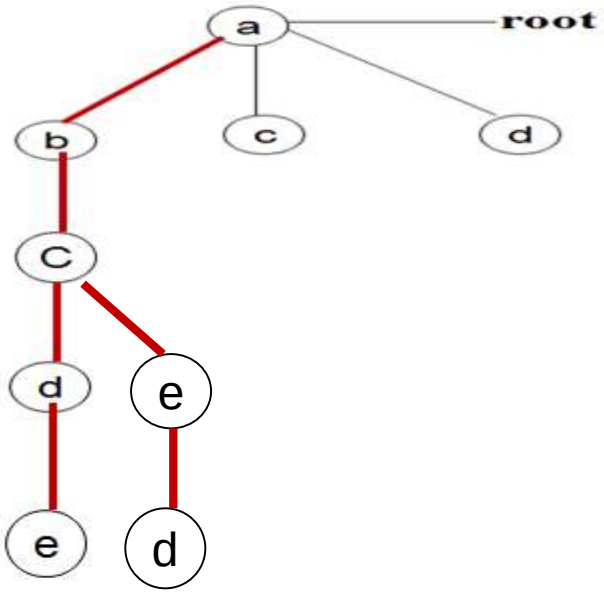
Hence we select the adjacent to c are b, d, e. the vertex b is already visited.

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



Dead end

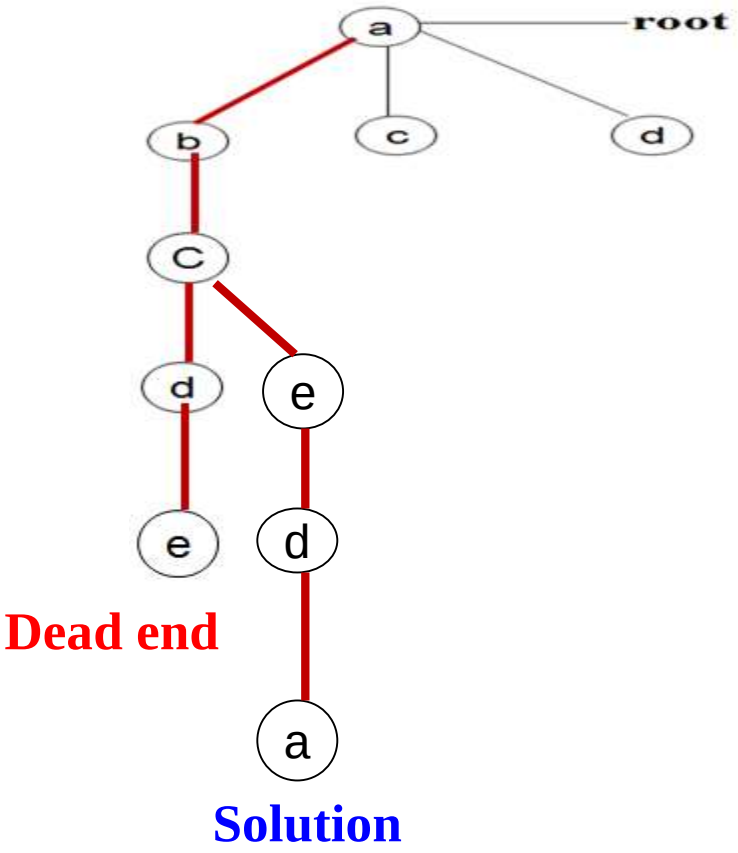
Adjacent to ‘e’ are (b, c, d), but b, c are already visited. The vertex ‘d’ is not visited . Thus we select ‘d’, which is adjacent to ‘e’.



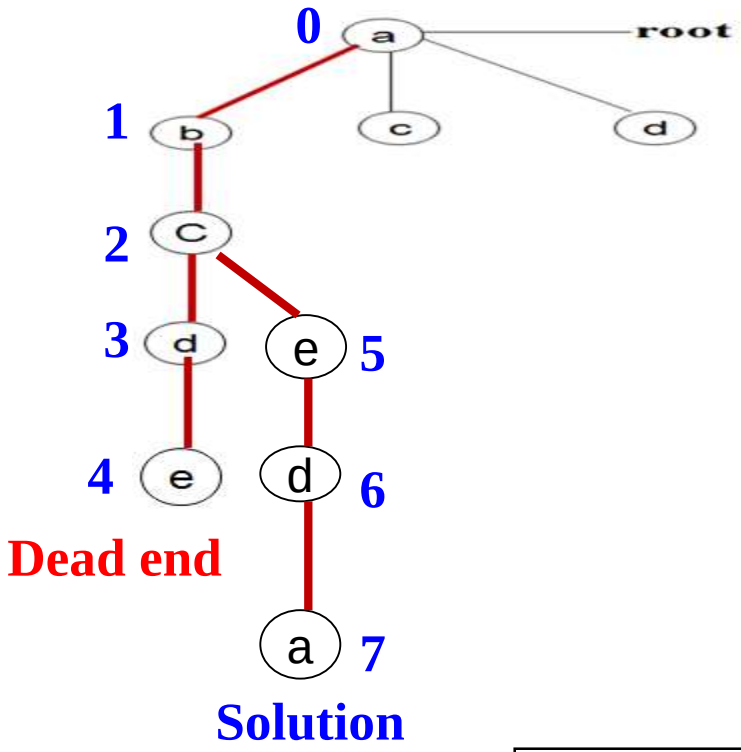
Dead end

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

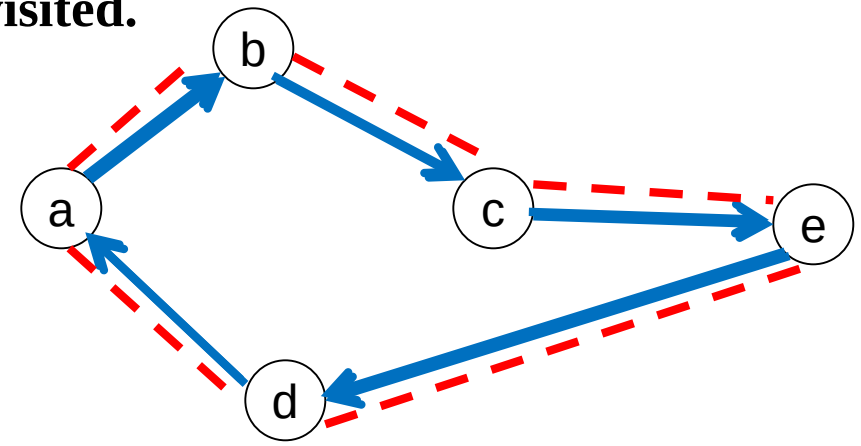
Hence we get the Hamiltonian cycle a-b-c-e-d-a.



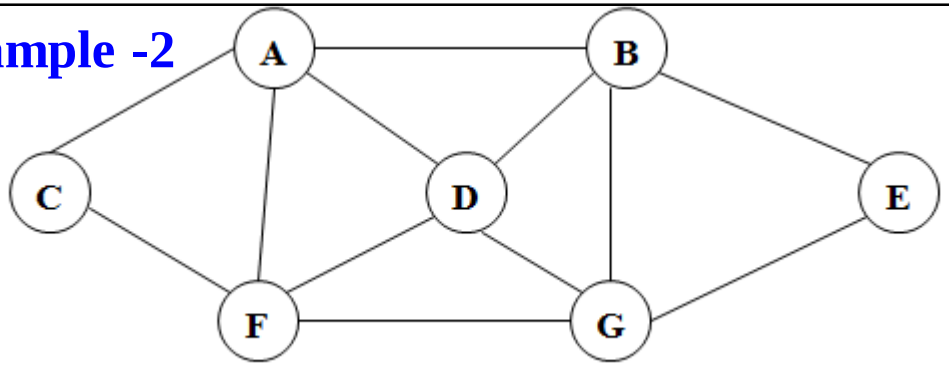
The final implicit tree for the Hamiltonian circuit is



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 );
}
```

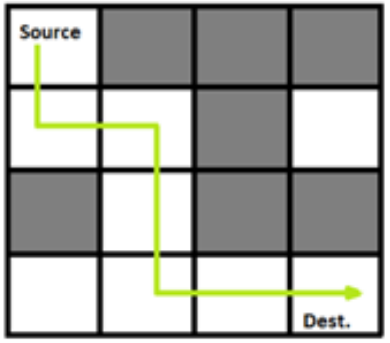
```
Algorithm NextValue( k )
{   //x[1:k-1] is a path of k-1 distinct vertices . If x[k]=0, then no vertex has as yet been assigned to x[k]. After
//execution, x[k] is assigned to the next highest numbered vertex which does not already appear in x[k-1] and is
//connected by an edge to x[k-1], otherwise 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 );
}
```

Mage Generation

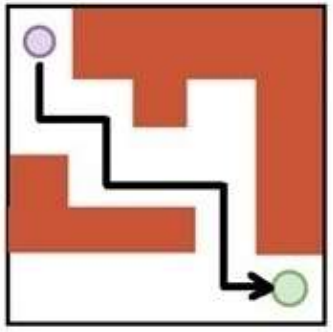
A N X N board is made in a way so that 1 or 0 can be placed in each cell where

- **1 is for valid path for moving towards exit or can be travel through it and**
- **0 is the closed path or blocked and cannot move to it while.**
- **Now by using backtracking, the path from source (0,0) to the exit destination at (N – 1, N – 1) to be find.**
- **The directions in which can move are 'U'(up), 'D'(down), 'L' (left), 'R' (right). Return the list of paths in lexicographically increasing order.**

Note: In a path, no cell can be visited more than one time. If the source cell is 0, cannot move to any other cell.



1	0	0	0	0
1	1	0	0	0
0	1	1	1	0
0	0	0	1	0
0	0	0	1	1



Procedure

- First, mark the starting cell as visited.
- Next, explore all directions to check if a valid cell exists or not.
- If there is a valid and unvisited cell is available, move to that cell and mark it as visited.
- If no valid cell is found, backtrack and check other cells until the exit point is reached.

```
1, 0, 0, 0
1, 1, 0, 1
1, 1, 0, 0
0, 1, 1, 1
```

DDRDRR
DRDDRR

Time Complexity: **O(3^(m*n)), because on every cell we have to try 3 different directions , as we will not check for the cell from which we have visited in the last move.**

Auxiliary Space: **O(m*n), Maximum Depth of the recursion tree(auxiliary space).**

Algorithm

```
Boolean Maze[n][n], Solution[n][n]

Boolean Solve_Maze()
{
    if (solve_Maze_Rec(0,0) = = false)
        return False;
    else
    {
        write(solution);
        return True;
    }
}
```

```
Boolean solve_Maze_Rec(int i, int j)
{
    if ((i = = N-1) and (j= = N-1)) then
    {
        Solution[i][j] = 1;
        return True;
    }
    if (is_Safe(i, j) = = True)) then
    {
        Solution[i][j] = 1;
        if (solve_Maze_Rec(i+1, j) = = True)) then
            return True;
        else if (solve_Maze_Rec(i , j+1) = = True)) then
            return True;
        Solution[i][j] = 0;
    }
    return False;
}
```

```
Boolean is_Safe(int i, int j)
{
    return((( i < N) and (j < N)) and (Maze[i][j] = = 1));
}
```

Example and State Space Tree