UNIT-II

BACKTRACKING

BACKTRACKING

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

i) Explicit constraints.

ii) Implicit constraints.

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

Some examples are,

$Xi \geq 0$ or Si = {all non-negative real nos.}

Xi =0 or 1 or Si={0,1}.

$Li \leq Xi \leq Ui$ or Si= {a: $Li \leq a \leq Ui$}

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

2) Implicit constraints: The implicit constraint determines which of the tuples in the solution space of I satisfy the criterion function. The implicit constrains describe the way in which the xi must relate to each other.

Example : In 4 – queen problem, The implicit constrains are no queens can be on the same column, same row and same diagnol.

CONTROL ABSTRACTION OF BACKTRACKING:

The control abstraction is also called as general method for backtracking is as follows. Let (X1, X2, X3, ……., Xi) be a path from the root to a node in a state space tree. Let (X1, X2, X3, ……., Xi) be the set of all possible values for xi+1 such that (X1, X2, X3, ……., Xi +1)is also a path to a problem state. We assume the existence of bounding function Bi+1 such that if Bi+1(X1, X2, X3, ……., Xi +1) is false for a path (X1, X2, X3, ……., Xi +1) from the root node to problem state, then the path cannot be extended to reach an answer node.

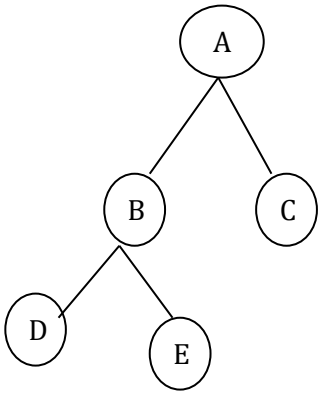
Algorithm : Backtracking (k)

```
{
    for ( each x[k] € T ( x[1], ….., x [ k – 1] ) do
    {
        if ( Bk ( x[1], x[2], …., x[k] ≠ 0 ) then
        {
            if ( x[1], x[2],……, x[k] is a path to an answer node ) then
                    write ( x[ 1 : k ] );
            if ( k < n) then
                Backtrack ( k + 1 );
        }
    }
}
```

BASIC TERMINOLOGY OF BACK TRACKING :

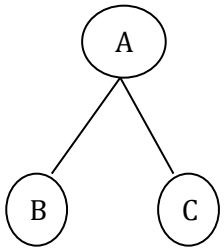
1 : Solution Space : All tuples that satisify the explicit constrains define a possible solution space for a particular instace 'I' of the problem.

Ex :



Here, ABD,ABE, AC are the tuples in solution space.

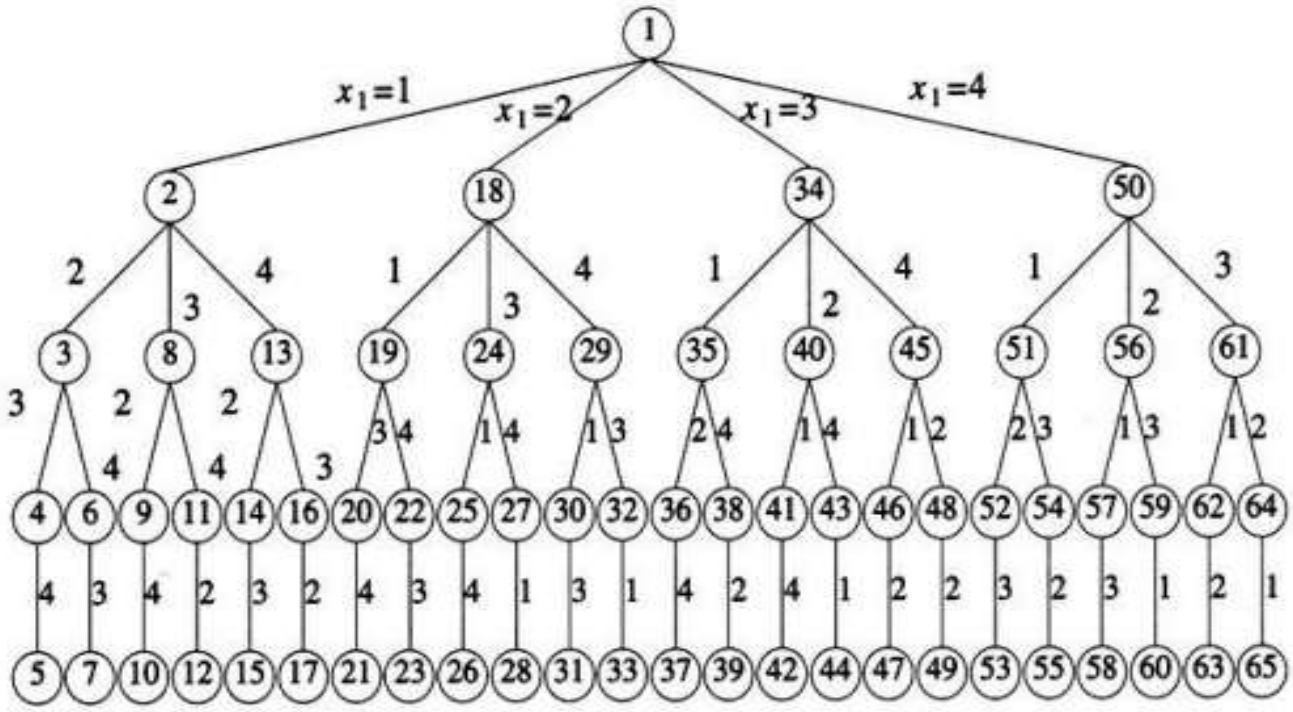
2 : Problem State : Each node in this tree defines a problem state.



So, the nodes A, B, C are problem state.

3 : State Space : State Space is the set of all paths from root node to other nodes.

Example : State space tree of a 4 – queen problem.



Tree organization of 4-queens solution space

4 : Solution States : Solution states are the problem states s for which the path from the root node to s defines a tuple in the solution space

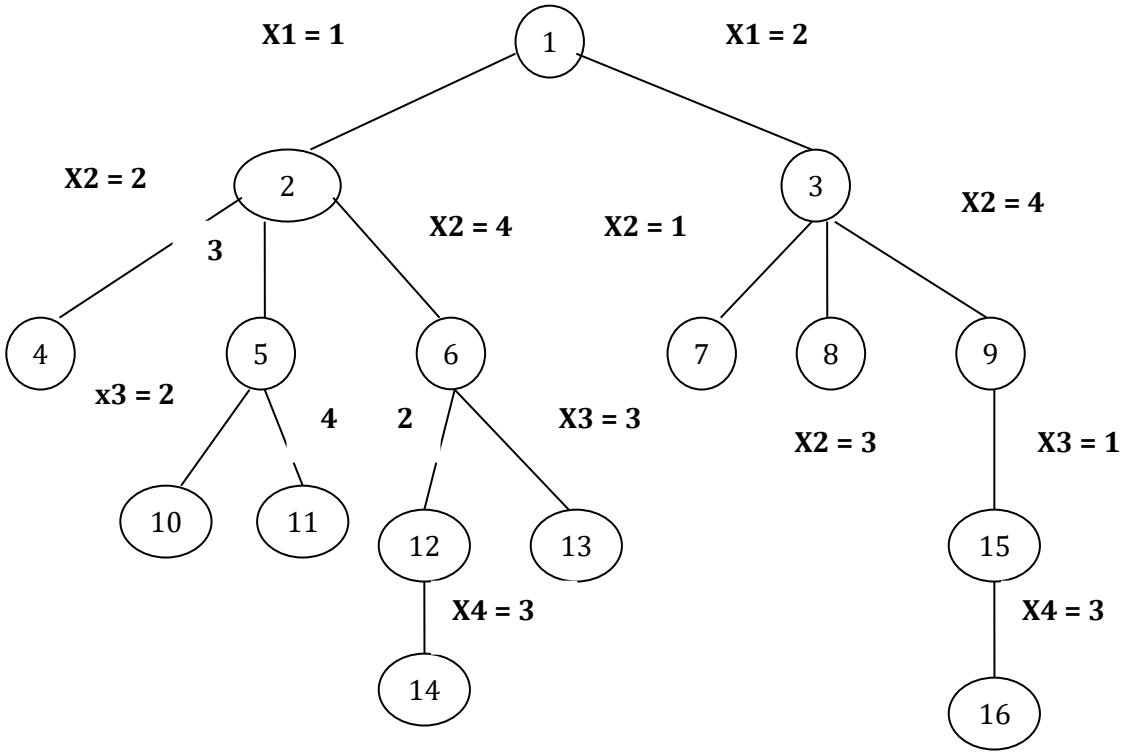
– In variable tuple size formulation tree, all nodes are solution states

– In fixed tuple size formulation tree, only the leaf nodes are solution states

– Partitioned into disjoint sub-solution spaces at each internal node

5 : Answer States : Answer states are those solution states s for which the path from the root to s defines a tuple that is a member of the set of solutions of the problem.

Example : The State Space Tree for the 4 – Queens problem is



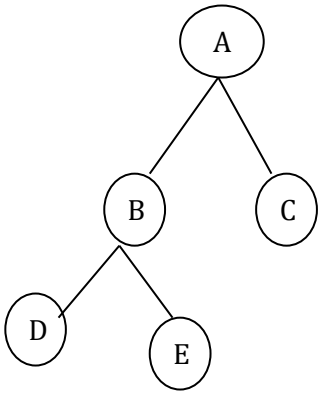
These Solution States represented in the form of tuples i.e, (1, 3, 9, 15, 16) are the solution states and Here 16 is Answer States

6: Live Node : A node which has been generated and all of whose children have not yet been generated is called Live Node.

Ex1 :

Here node A is called Live node since the children of node A have not yet been generated.

Ex 2 :



Here, nodes D, E, C are live nodes because the children of these nodes are not yet been generated.

10 : Bounding Function : is a function used to kill live nodes without generating all their children.

Applications of Backtracking :

1 : N – Queen Problems.

2 : SumOfSubset Problem.

3 : Graph Coloring Problems.

4 : Hamiltoniam Graph Problem.

1 : N QUEENS PROBLEMS (4 – QUEENS AND 8 – QUEENS PROBLEM) : Consider an n * n chessboard. Let there are N queens. These n – quens are to be placed on the n * n chessboard so that no two queens are on the same column, same row or same diagnol.

Algorithm : NQueens(k , n)

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

Algorithm place (k, i)

```
{
  for j =1 to k – 1 do
  {
     If(X[k] = i) then
        return false;
    else
        retrn true;
}
```

4 – QUEENS PROBLEM : Consider a 4 * 4 chessboard let there are 4 – queens. These 4 – queens are to be placed on the 4 * 4 chessboard . So that no two queens are on the same column, same row or same diagnol position.The explicit constrains are 4 – queens are to be placed on 4 * 4 chessboard in 4 ways. The implicit constrains are no two queens are in the same row or column or diagnol position.

Let { x1, x2, x3, x4 } be the solution vector where xi, column number on which the queen i is placed.

Step 1 : First queen Q1 is placed in first row and first column.

Q1			

Step 2 : The second queen should not b placed in first row and second column. It should be placed in second row and in second, third or fourth column. If we place in second column, both will be in same diagnol, so place it in third column.

Q1			
*	*	Q2	

Step 3 : We are unble to place Q3 in third row .

Q1			
*	*	Q2	
*	*	*	*

Then go back to Q2 and place it somewhere else and then place Q3 in 3rd row, 2nd column.

Q1			
			Q2

Q1			
			Q2

	Q3		

Step 4 : The fourth queen should be placed in 4^{th} row and 3^{rd} column but there will be a diagnol attack from Q3. so go back, remove Q3 and place it in the next column. But it is not possible, so move back to Q2 and remove it to next column but it is not possible. So go back to Q1 and move it to next column. It can be shown as follows.

	Q1		

	Q1		
			Q2

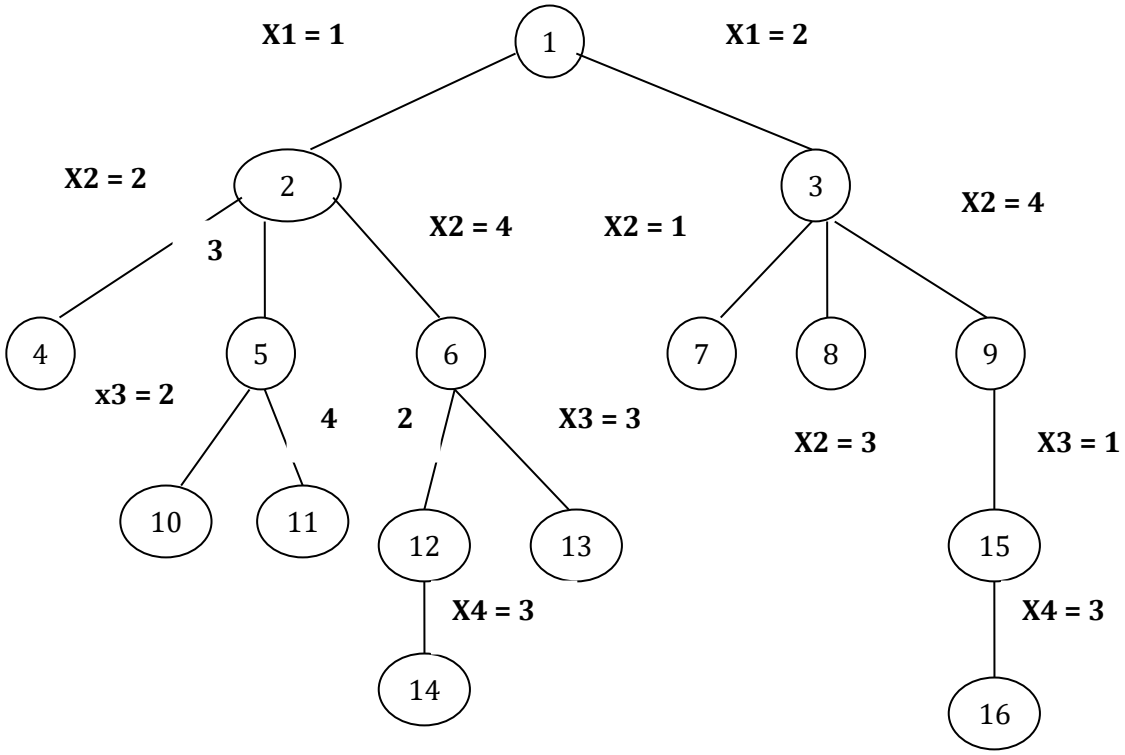
	Q1		
			Q2
Q3			

	Q1		
			Q2

Q3			
		Q4	

Hence the solution to 4 – queen's problem is x1 = 2 , x2 = 4, x3 = 1, x4 = 3. i.e, first queen is placed in 2nd column, second queen is placed in 4th column and third queen is placed in first column and fourth queen is placed in third column.

The State Space Tree for the 4 – Queens problem is



2.SUM OF SUBSETS : We are given n distinct positive numbers (usually called weights) and we desire to find all combinations of these numbers whose sums are m. This is called Sum of Subsets problem.

Example 1 : if given set S = (1, 2, 3, 4) and M = 5, then

there exists sets **S '** = (3 , 2) and S' = (1,4) whose sum is equal to M

Example 2 : if a given set S = (1, 3, 5) and M = 7, then

No subset occurs for which the sum is equal to M = 7.

We could formulate Sum Of Subsets Problem using either fixed or variable sized tuples. Now we consider a backtracking solution using the fixed sized tuple. In this case the element x_i of the solution vector is either zero or one depending on whether the weight w_i is included or not.

The children of any node are easily generated. For a node at level I the left child corresponds to $x_i = 1$ and the right child corresponds to $x_i = 0$.

Algorithm : SumOfSub(s, k, r)

```
{
        x[k] = 1;
        if ( s + w[k] = m) then
                write ( x[1 : k]) ;
        else if ( s + w[k] + w[k + 1] ≤ m ) then
                SunOfSub(s + w[k], k + 1, r – w[k]);
        if ( ( s + r – w[k] ≥ m) and ( s + w[k+1] ≤ m) then
        {
                x[k] = 0;
                SumOfSub( s, k+1, r – w[k]);
        }
}
```

Example1: Let n=3, w = {2, 4, 6} and m = 6, Find all possible subsets of w that sum to m.

Solution : n = 3

M=6

w1 = 2, w2=4,w3=6

Initially three variables

S = 0

k = 1

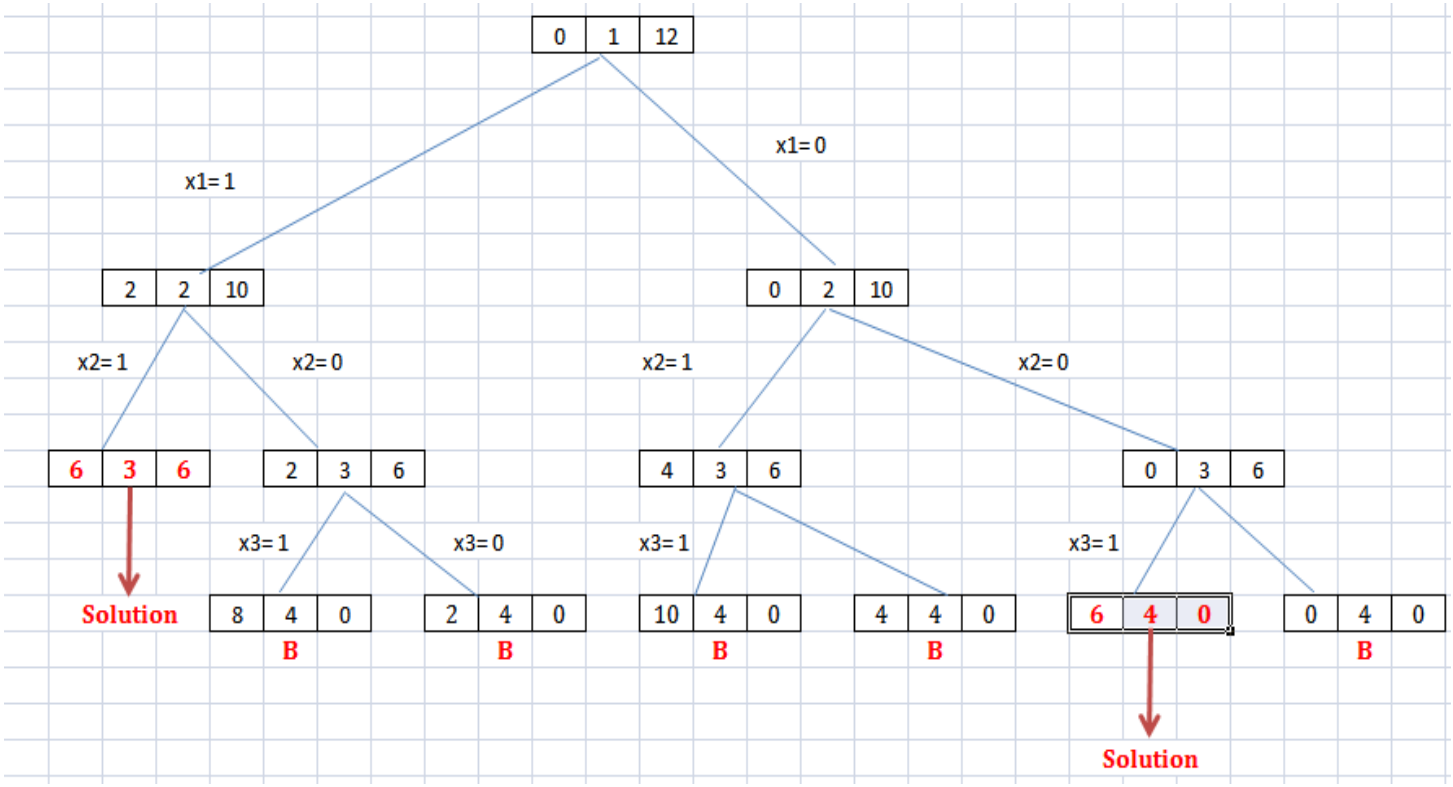
r = (2 + 4 + 6) = 12

To calculate Left child and Right child

x_i = 1 (Left Child) = s + w[k],k+1,r-w[k]

x_i = 0 (Right Child) = s ,k+1,r-w[k]

State space tree generated by Sum of Subsets Problem



There are Two answer states in the state space tree diagram. The answer states are denoted by A,B. There are 2 solutions

A = { 1, 1, 0 } = { 2 + 4 + 0 } = 6

B = { 0, 0, 6 } = { 0 + 0 + 6 } = 6

Example 2 : Let n=6, w ={ 5,10,12,13,15,18 } and m = 30, Find all possible subsets of w that sum to m.

Solution : n = 6, M=30

w1 = 5, w2=10, w3=12, w4=13, w5=15, w6=18

Initially three variables

S = 0

k = 1

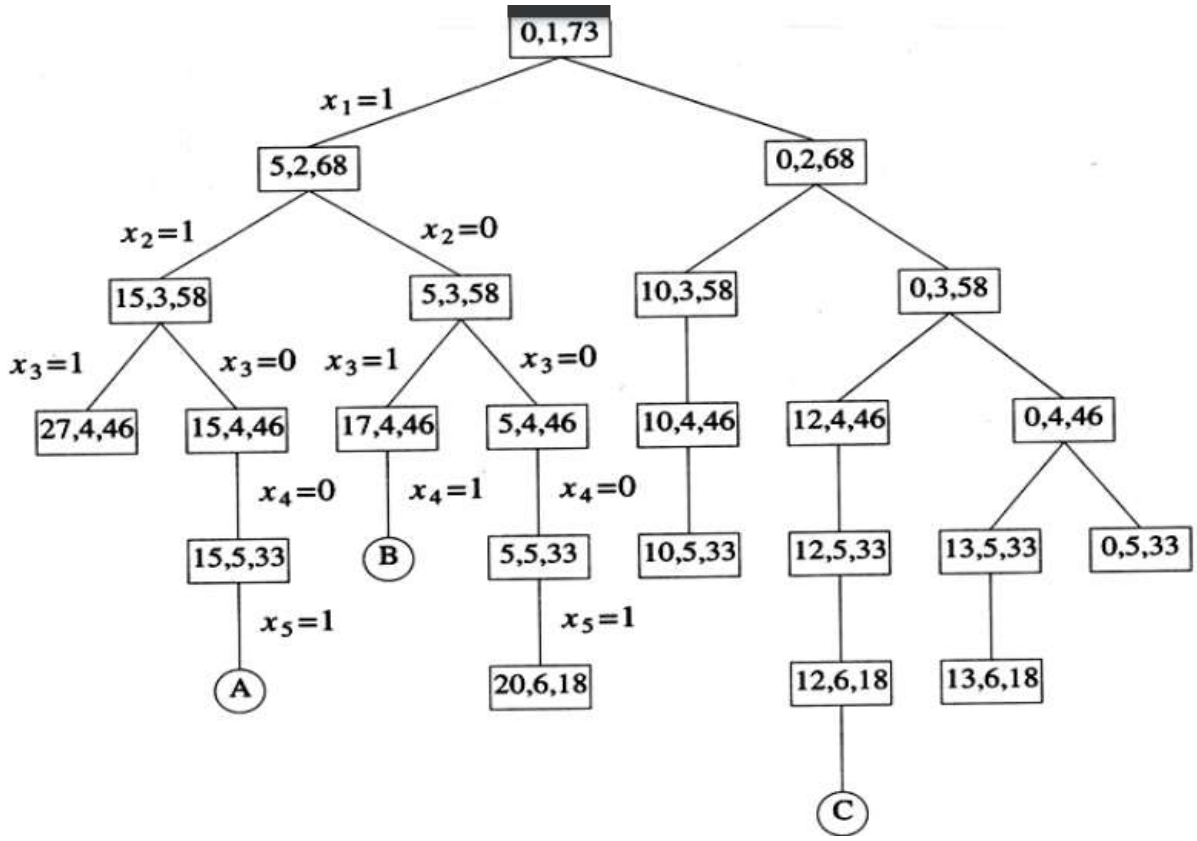
r = (5+10+12+13+15+18) = 73

To calculate Left child and Right child

x_i = 1 (Left Child) = s + w[k],k+1,r-w[k]

x_i = 0 (Right Child) = s ,k+1,r-w[k]

State space tree generated by Sum of Subsets Problem



There are Three answer states in the state space tree diagram. The answer states are denoted by A,B,C. There are 3 solutions

A = { 1, 1, 0, 0, 1, 0 } = { 5 + 10 + 0 + 0 + 15 + 0 } = 30

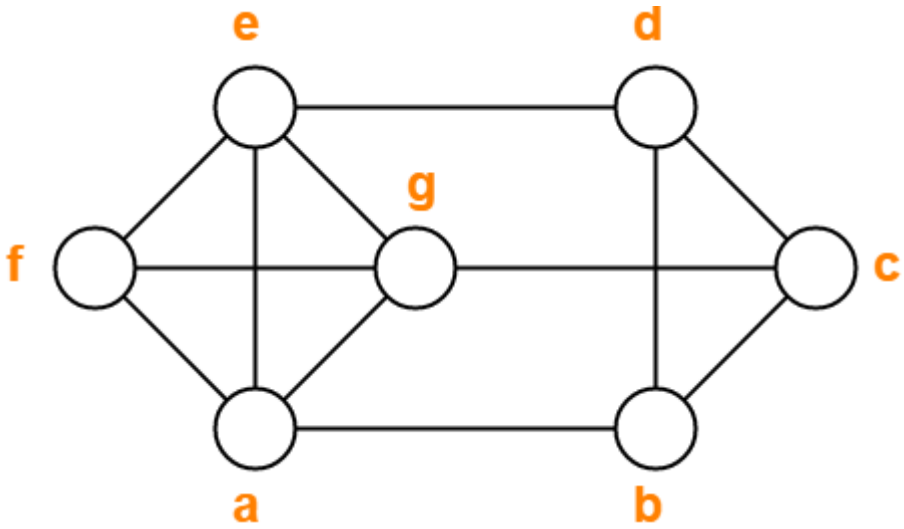
B = { 1, 0, 1, 1, 0, 0 } = { 5 + 0 + 12 + 13 + 0 + 0 } = 30

C= { 0, 0, 1, 0, 0, 1 } = { 0 + 0 + 12 + 0 + 0 + 18 } = 30

GRAPH COLOURING PROBLEM :

- Let G be a graph and m be a given positive integer.
- The graph coloring problem is to discover whether the nodes of G can be colored in such a way that no two adjacent nodes / vertices have the same color yet only m colors are used.
- This graph coloring problem is also known as m-colorability decision problem
- The m-Coloring optimization problem asks for the smallest integer m for which the graph G can be colored. This integer is referred to as the chromatic number of the graph.
- **Example**, the graph can be colored with 4 colors. Hence the chromatic number of the graph is 4.
- Graph coloring problem is a NP Complete problem.
- If d is the degree of the given graph, then it can be colored with d+1 colors.

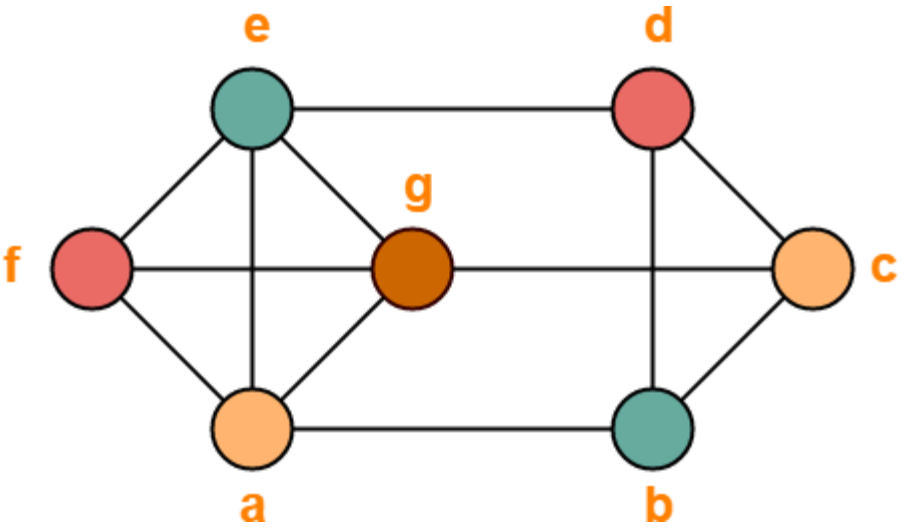
Example : Consider the following graph



From here,

- Minimum number of colors used to color the given graph are 4.
- Therefore, Chromatic Number of the given graph = 4.

The given graph may be properly colored using 4 colors as shown below-



Algorithm for Graph Coloring Problem

```
Algorithm mColoring(k)
{
        repeat
        {
                NextValue(k);
                if(x[k]=0) then
                        return;
                if(k=n) then
                        write(x[1:n]);
                else
                        mColoring(k+1);
        }
        until false;
}
Algorithm NextValue(k)
{
        repeat
        {
                x[k] = (x[k]+1)mod(m+1);
                if(x[k]=0) then
                        return;
                for j =1 to n do
                {
                        if((G[k,j] ≠ 0) and (x[k] = x[j])) then
                                break;
                }
                if(j=n+1) then
                        return;
        }
        until(false);
}
```

Graph Coloring Applications-

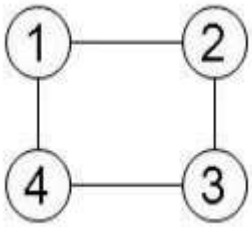
Some important applications of graph coloring are as follows-

- Map Coloring
- Scheduling the tasks
- Preparing Time Table
- Assignment
- Conflict Resolution
- Sudoku

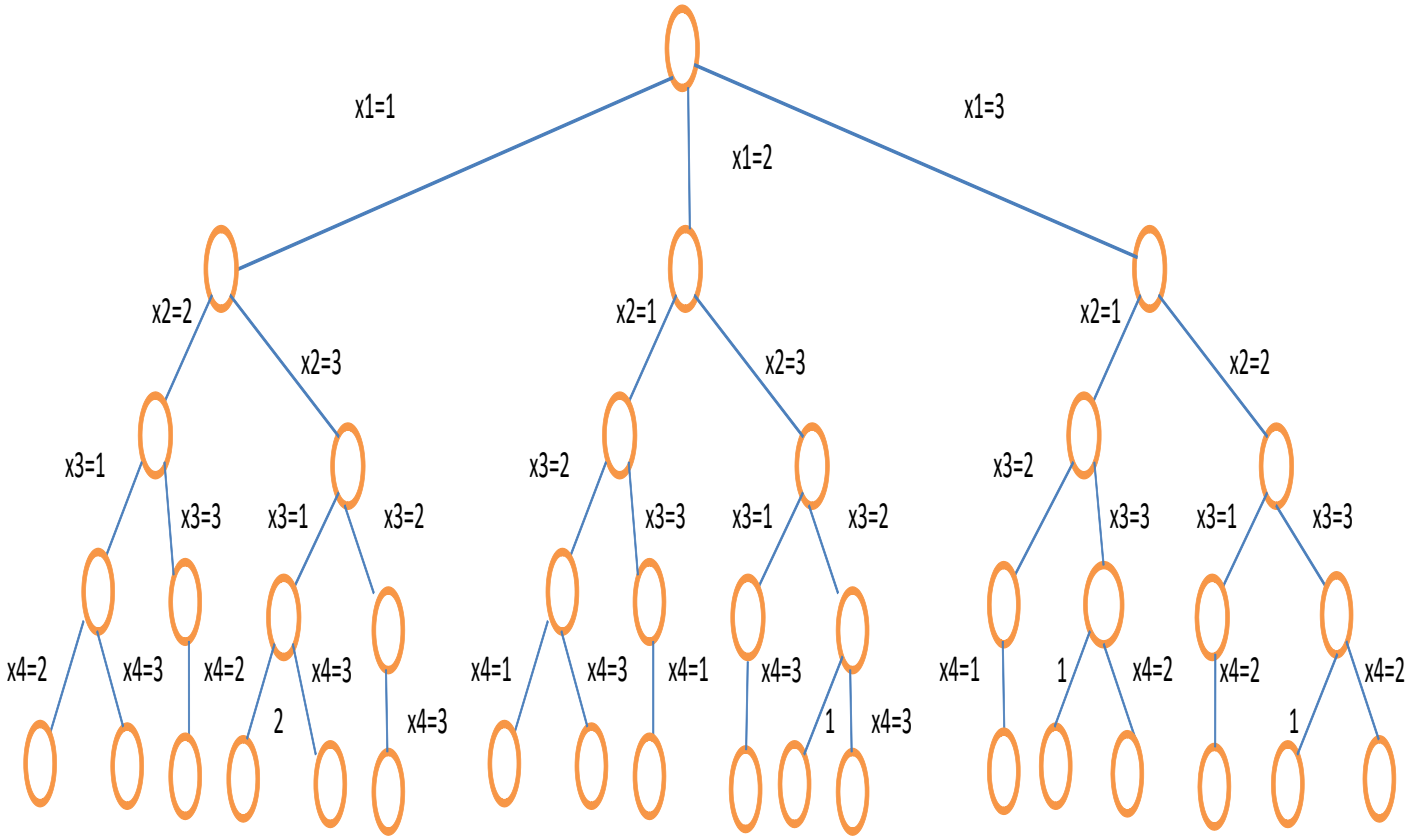
Time Complexity: **O(mV)**.

Space Complexity: **O(V)**

Example : **A 4-node graph and all possible 3-colorings**



Solution : **State Space tree for graph coloring problem (n=4, m=3)**



Here The Solutions are

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

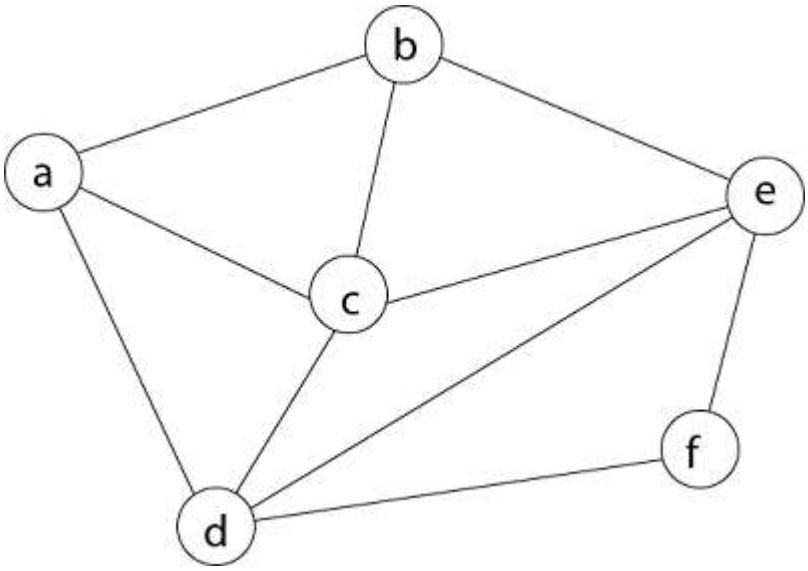
HAMILTONIAN GRAPH

Definition : A Hamiltonian graph is the directed or undirected graph containing a Hamiltonian cycle. The Hamiltonian cycle is the cycle that traverses all the vertices of the given graph G exactly once and then ends at the starting vertex.

- Given a graph G = (V, E) we have to find the Hamiltonian Circuit using Backtracking approach.
- We start our search from any arbitrary vertex say 'a.' This vertex 'a' becomes the root of our implicit tree.
- The first element of our partial solution is the first intermediate vertex of the Hamiltonian Cycle that is to be constructed.
- The next adjacent vertex is selected by alphabetical order. If at any stage any arbitrary vertex makes a cycle with any vertex other than vertex 'a' then we say that dead end is reached.
- In this case, we backtrack one step, and again the search begins by selecting another vertex and backtrack the element from the partial solution must be removed.
- The search using backtracking is successful if a Hamiltonian Cycle is obtained

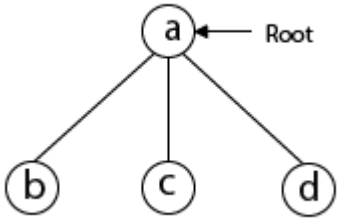
```
Algorithm Hamiltonian(k)
{
        repeat
        {
                NextValue(k);
                if(x[k]=0) then
                        return;
                if(k=n) then
                        write(x[1:n]);
                else
                        Hamiltonian (k+1);
        }
        until false;
}
Algorithm NextValue(k)
{
        repeat
        {
                x[k] = (x[k]+1)mod(n+1);
                if(x[k]=0) then return;
                if(G[x[k-1],x[k]] ≠0) then
                {
                        for j =1 to k-1 do
                                if(x[j] = x[k])) then
                                break;
                        if(j=k) then
                                if((k<n) or ((k=n) and G[x[n], x[1]] ≠0)) then
                                        return;
                }
        }
        until(false);
}
```

Example : The following graph illustrates the presence of the Hamiltonian cycle in a graph G.

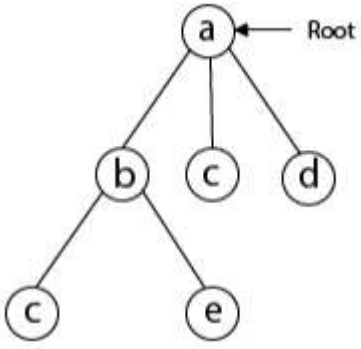


Step 1: Firstly, we start our search with vertex 'a.' this vertex 'a' becomes the root of our implicit tree.

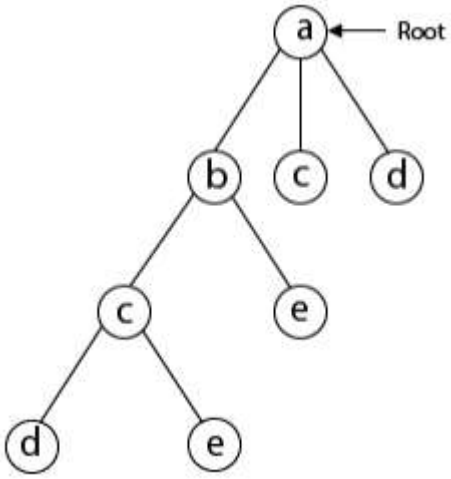
Step 2: Next, The vertex adjacent to 'a' is b, c and d. it comes first in lexicographical order (b, c, d).



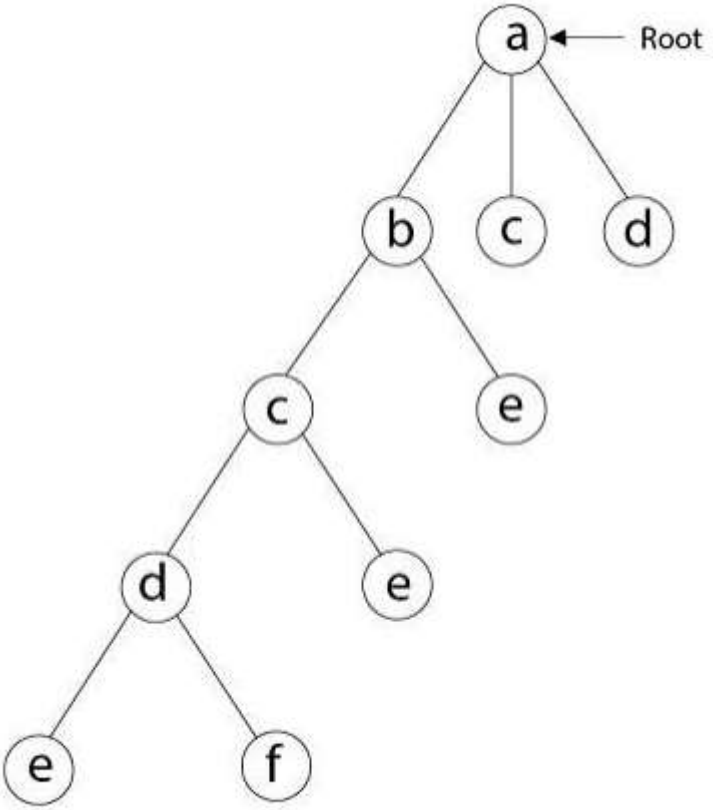
Step 3: Next, The vertex adjacent to 'b' is a, c and e. but ‘ a’ is already visited.



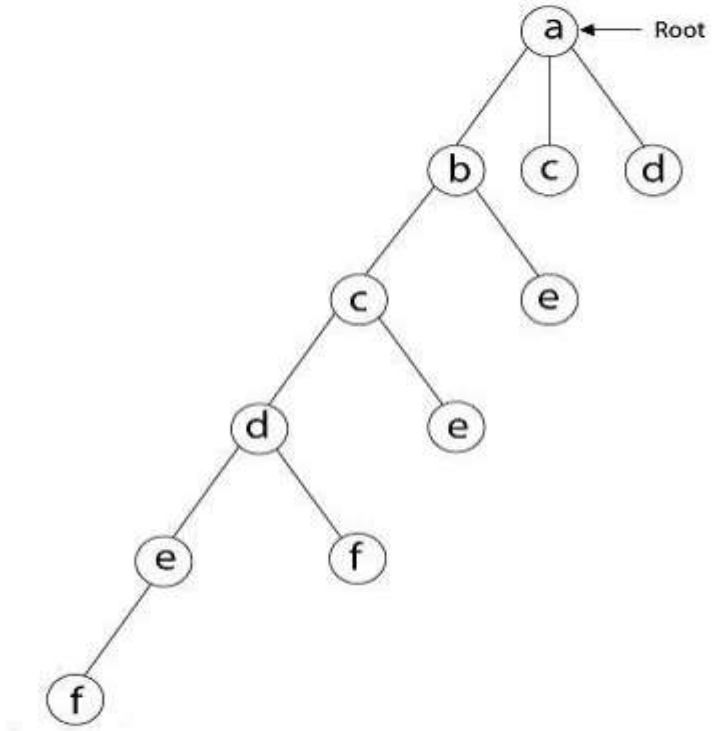
Step 4 : Next, The vertex adjacent to 'c' is a, b, d and e. but ‘ a’ and ‘b’ are already visited.



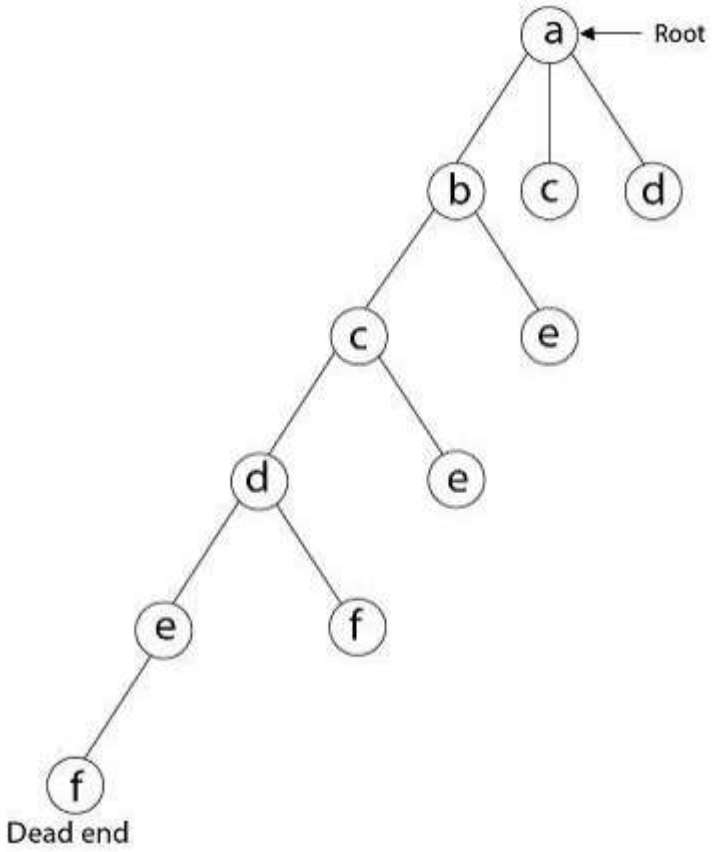
Step 5: Next, The vertex adjacent to 'd' is a, c, e and f. but ‘ a’ and ‘c’ are already visited.



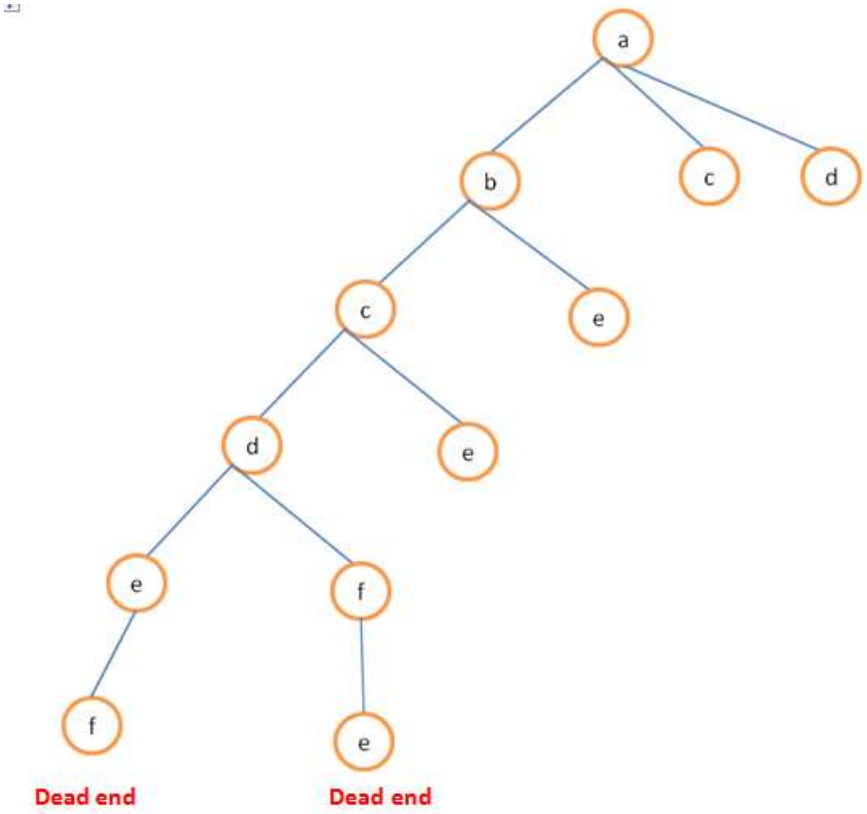
Step 6 : Next, The vertex adjacent to 'e' is b, c and f. but ‘ b’ and ‘c’ are already visited.



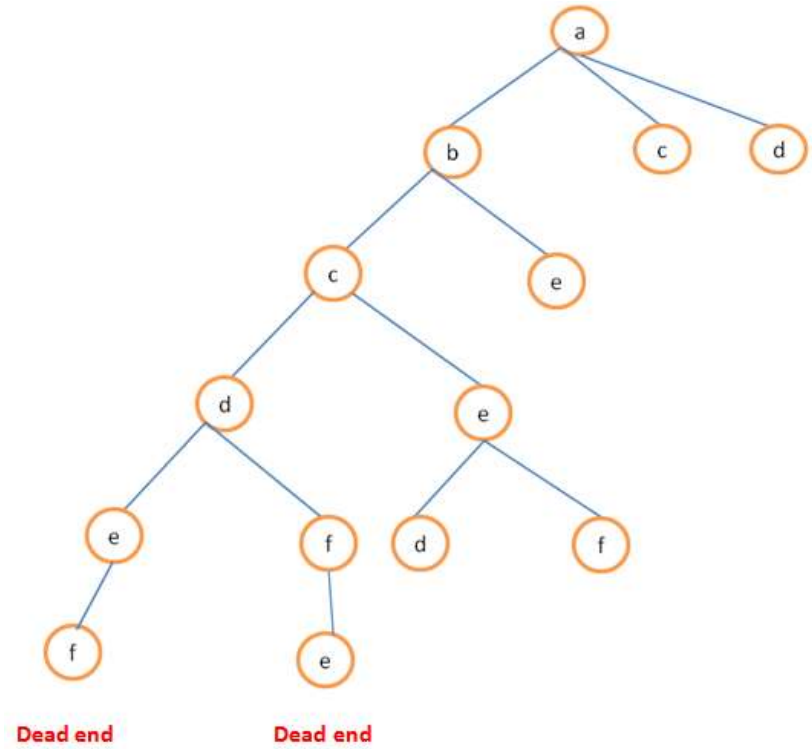
Step 7 : Next, The vertex adjacent to 'f' is d and e, but ‘d’ and ‘e’ are already visited. Thus, we get the dead end, and we backtrack one step



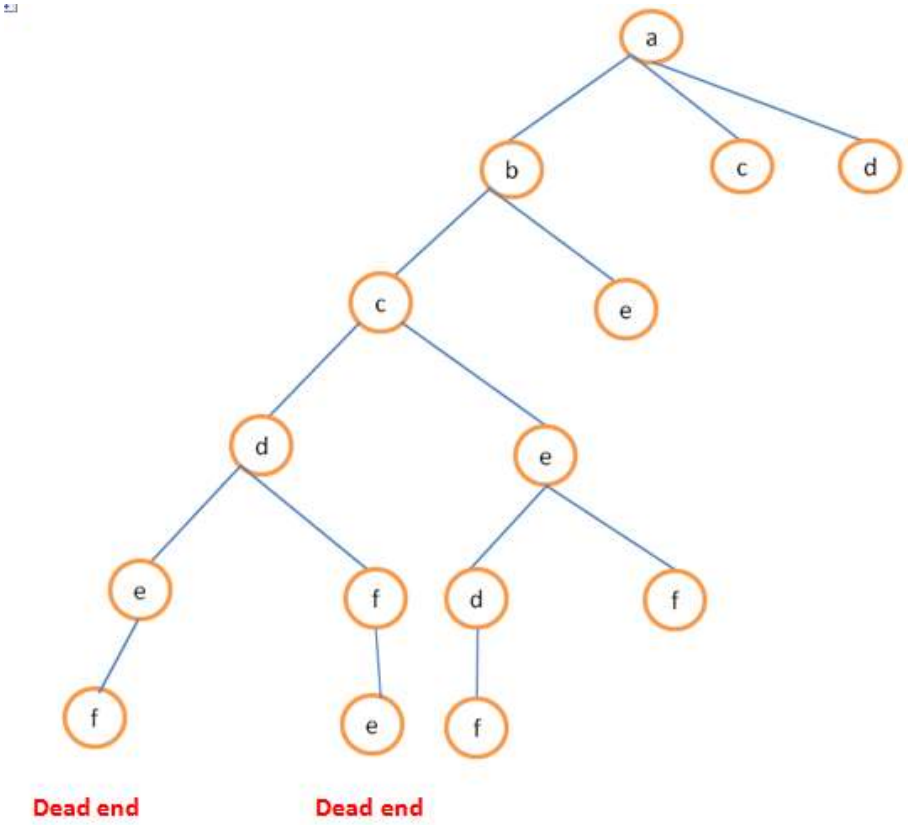
Step 8 : we backtrack one step , the vertex adjacent to 'e' is b, c, d, and f from which vertex 'f' has already been checked, and b, c, d have already visited. So, again we backtrack one step. Now, the vertex adjacent to d are e, f from which e has already been checked, and adjacent of 'f' are d and e. The vertex 'd' has already visited. If 'e' vertex, revisited them we get a dead end. So again we backtrack one step.



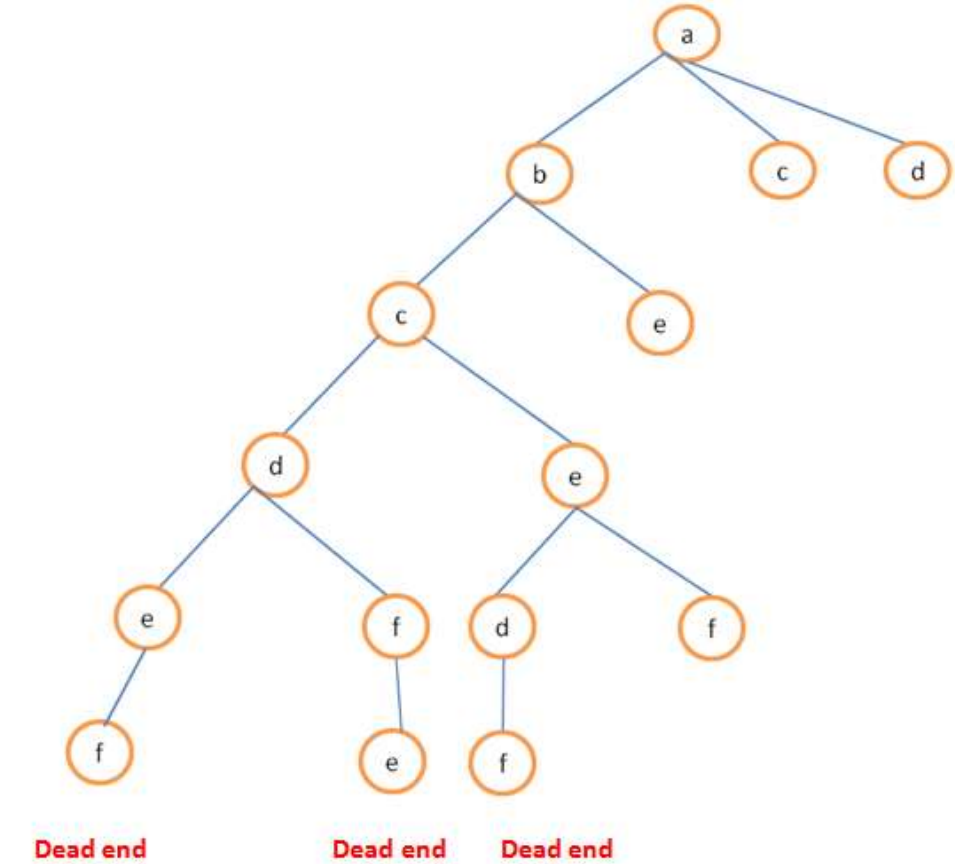
Step 9 : we backtrack one step , Next, The vertex adjacent to 'e' is b,c,d and f, but 'b' and 'c' are already visited.



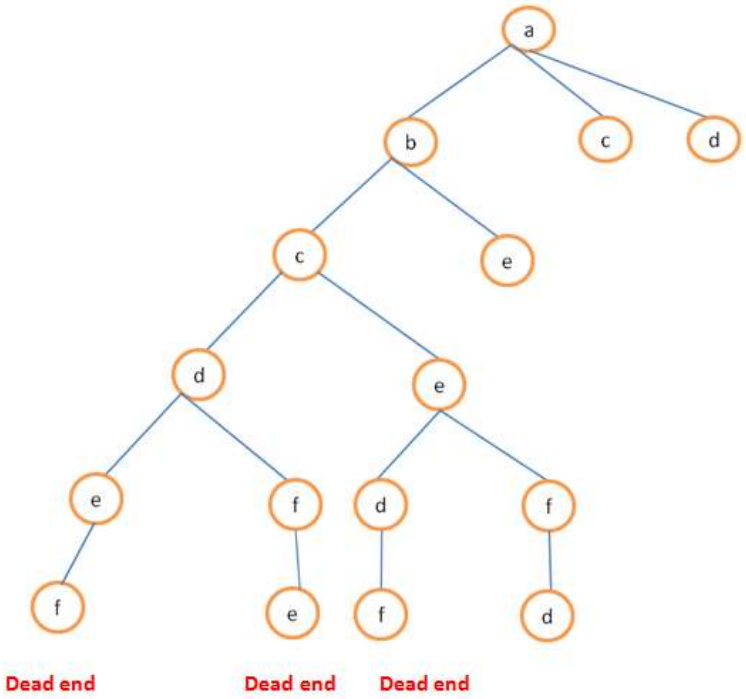
Step 9 : Next, The vertex adjacent to 'd' is a,c,e and f, but 'a','c' and 'e' are already visited.



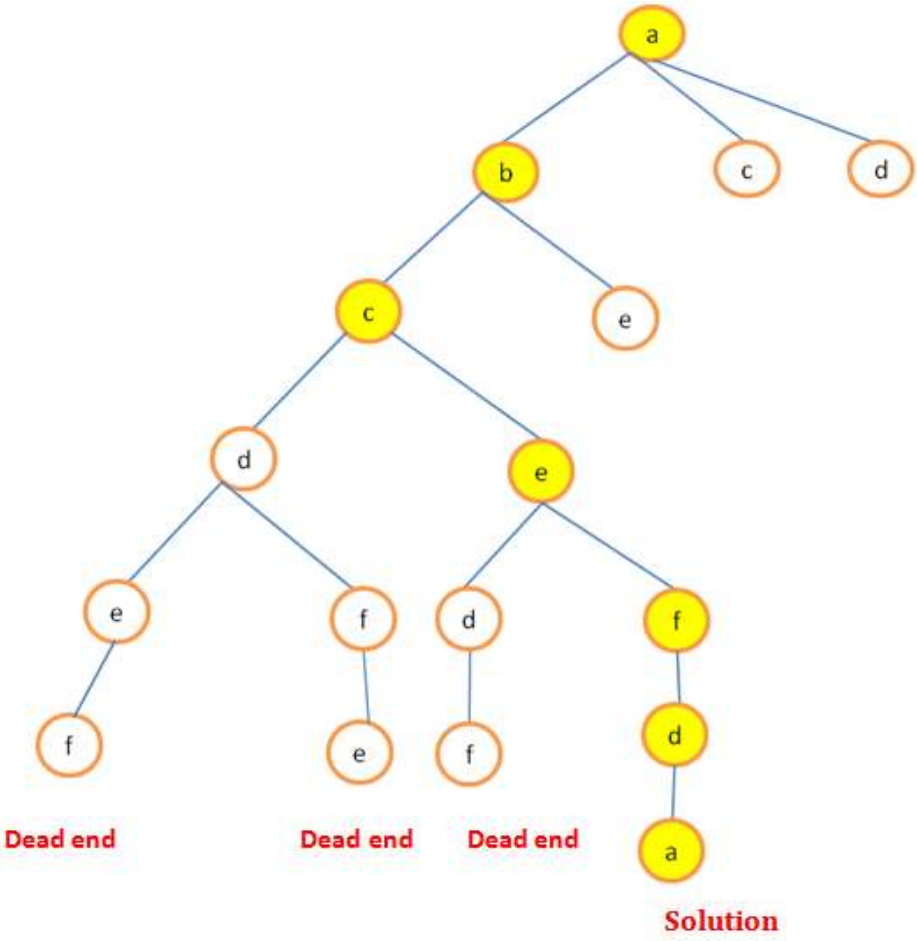
Step10 : Next, The vertex adjacent to 'f' is d,e but 'd',and 'e' are already visited. Thus, we get the dead end



Step11 : Next, The vertex adjacent to 'f' is d,e but 'e' are already visited.



Step12 : Next, The vertex adjacent to 'd' is d,e but 'e' are already visited.



The Solution is : a – b – c – e – f – d – a