Unit IV

BACKTRACKING

Introduction :-

In case of greedy and dynamic programming techniques, we will use Brute force approach. It means we will evaluate all possible sols, among which, we select one sol as optimal sol. In backtracking technique, we will get same optimal sol with a less no of steps. So by using backtracking technique, we will solve problems in an efficient way, which compared to other methods like greedy method and dynamic prg'ing. In this we will use bounding fns (criterion fn), implicit and explicit constraints. While explaining the general method of tracking technique, there we will see implicit and explicit constraints. The major advantage of backtracking method is, if a partial sol $(x_1, x_2, \ldots x_i)$ can't lead to optimal sol then $(x_{i+1} \ldots x_n)$ sol may be ignored entirely.

Explicit constraints : These are rules which restrict each x_i to take on values only from a given set.

Eg: (1) In Knapsack prob, the explicit constraints are,
(i) $x_i \geq 0$ or 1
(ii) $0 \leq x_i \leq 1$.

(2) In 4-queens prob, 4-queens can be placed in 4×4 chessboard in 4^4 ways.

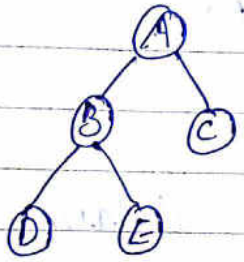
Implicit Constraints : These are rules which determine which of the tuples in the sol space satisfy the criterion fn.

Eg: In 4- queens prob, the implicit constraints are no two queens can be on the same col, same row and same diagonal.

Criterion Function: It is a fn $P(x_1, x_2, \ldots x_n)$ which needs to be maximized for a given problem.

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

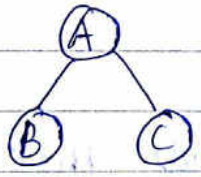
Eg:



ABD, ABE, AC are the tuples in sol space.

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

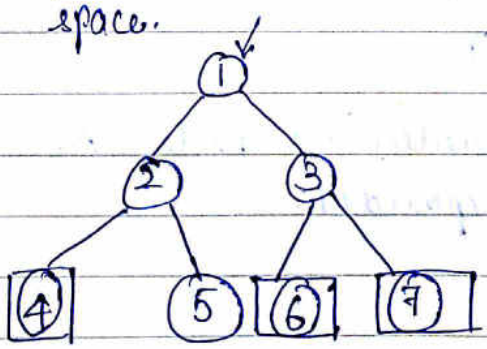
Eg:



∴ A, B, C are nodes in prob state.

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

Eg:

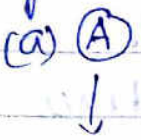


Here, square nodes indicates sol. For the above sol space, there exists 3 sol states. These sol states represented in the form of tuples ie (1,2,4), (1,3,6) and (1,3,7).

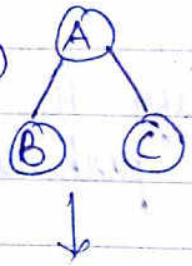
State Space Tree: If we represent sol space in the form of a tree then the tree is refered as the state space tree.

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

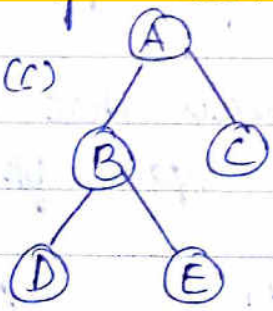
Eg:

(a) 

live node

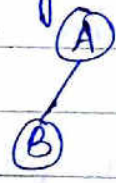
(b) 

B, C are live nodes

(c) 

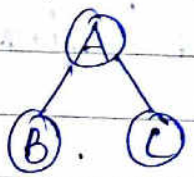
→ D, E, C are live nodes.

E-node:- The live nodes whose children are currently being generated is called the E-node.

Eg: 

E node Here (A) is E node.

Dead Node: It is a 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 deadnodes, ∵ node A's children generated & node B, C are not expanded.

General Method (Control Abstraction) :- (8 T)

In this technique, we search for the set of sols or optimal sol which satisfies some constraints. The desired sol is expressed as an n-tuple $(x_1, x_2, \dots x_n)$ where x_i is chosen from some finite set S_i. The sol maximizes or minimizes or satisfies a criterion fn (objective fn) $c(x_1, x_2, \dots x_n)$. The basic idea of backtracking 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 alg is that we can realize the fact that partial vector generated does not lead to an optimal sol. In such a situation that vector can be ignored. The backtracking alg determine the sol by systematically searching the sol space for the given prob. All sols using backtracking are required to satisfy a complex set of constraints. These constraints may be explicit or implicit.

- Algorithm Backtrack(n)
{
  K ← 1;
  while(K ≠ 0) do
  {
    if there remains an untried x[K] ∈ T(x[1], x[2], x[K-1]) and
    B_K(x[1], x[2], ... x[K]) then
    {
      if (x[1], x[2], ... x[K]) is a path to an answer node then write(x[1:K]);
      K ← K+1;
    }
    else K ← K-1;
  }
}

Applications:

I. 8-Queens Problem:-

Consider a 8×8 chessboard. Let there are 8 queens. The objective is to place these 8 queens on the board so that no two queens are on the same row or same col or same diagonal.

Let $(x_1, x_2, \ldots x_8)$ represent a sol in which x_i is the col number on which the queen i is placed in the same col.

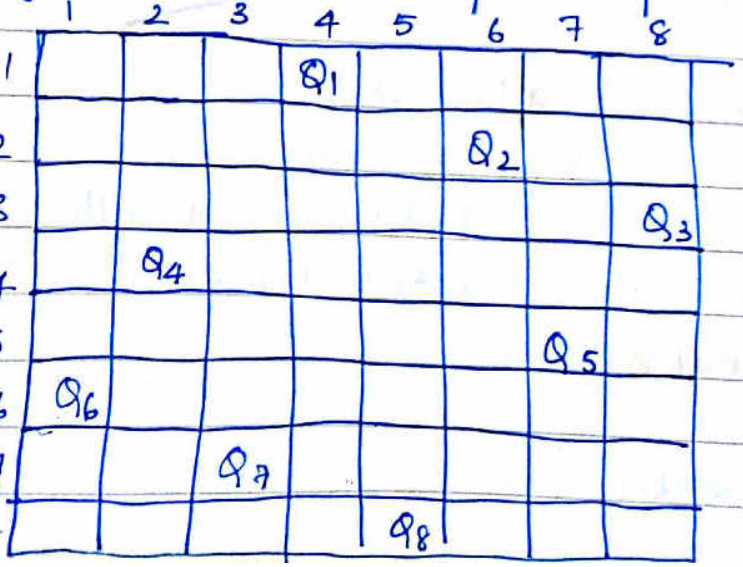
Let A[1:8, 1:8] be the two dimensional array representating the squares of chess board. The queens are numbered 1 through 8. Each queen must be on a different row.

The explicit constraint is 8-queens are to be placed in 8×8 chess board in 8^8 ways. We reduce the sol space of explicit constraints by applying the implicit constraints.

The implicit constraints are no two queens are in the same row, or col or diagonal.

Suppose (i, j) and (k, l) are the two positions for two queens. They are on the same diagonal iff $|j-l| = |i-k|$.

A typical sol to 8-queens prob is



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

Time Complexity: The sol space tree of 8-queens problem contains 8^8 tuple. After imposing implicit constraints, the size of sol space is reduced to 8! tuples. Hence the time complexity is O(8!). For n-queens problem, it is O(n!).

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

Algorithm place (k)
{ // a queen is placed in kth row & ith col return
  // true otherwise false.
  for j ← 1 to k-1 do
  { if ((x[j] ← i)   // Two in the same column
    or (ABS(x[j]-i) = ABS(j-k)) // or in the same
                                // diagonal
    then return false;
    else
      return true;
  }
```

1. Sum of Subsets Problem:-

If there are n positive numbers in a given set, then the desire sol^n is to find all possible subsets of this set, the contents of which to add onto a predefined value M. In otherwords, let there be n elts given by the set W = (W1, W2, ... Wn) then find out all the subsets from whose sum is M.

The sum of subset is based on fixed size tuple. Let us draw a tree structure for fixed tuple size formulation.

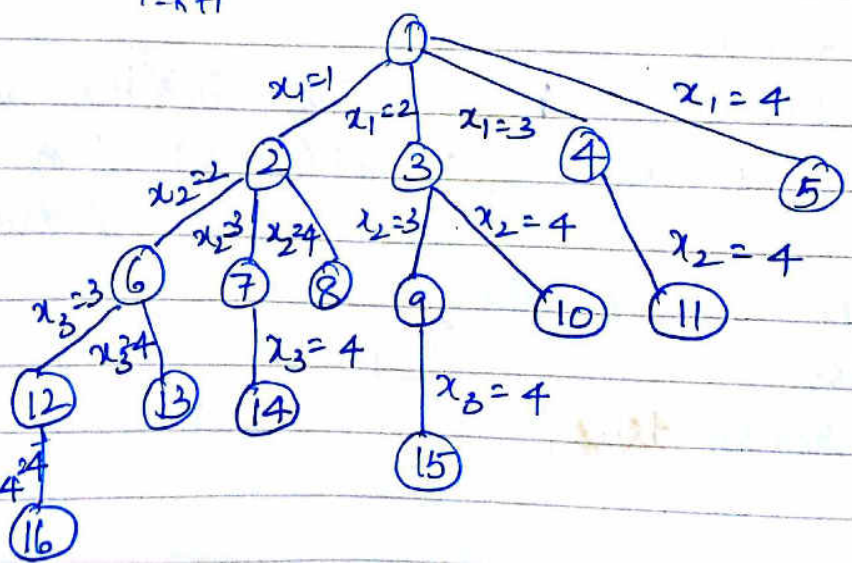
All paths from root to a leaf node define a sol space. The left subtree of the root defines all subsets containing w_1 and the right subtree defines all subsets not containing W_1 and so on.

Bounding fn is a fn which is used to kill live nodes without generating all their children.

The choice for bounding fn is

$B_K(a_1, \ldots a_K)$ is true iff,

$$\sum_{i=1}^{K} w_i a_i + \sum_{i=K+1}^{n} w_i \geq M \longrightarrow (1)$$



$a_1, \ldots a_k$ cannot lead to answer if eq ① is not satisfied.

For strengthening bounding fn, we will sort w_i in non-decreasing order.
In this case $x_1, \ldots x_k$ cannot lead to an answer node if,

$$\sum_{i=1} w_i a_i + w_{k+1} > m.$$

$B_k(a_1, a_2, \ldots a_k) = \text{true}$

iff,

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

and

$$\sum_{i=1}^{k} w_i a_i + w_{k+1} \leq M.$$

Let S be a set of elts and N is the expected sum of subsets. Then,

Step 1: Start with an empty set.
2: Add to the subset next elt from the list.
3: If the subset is having sum N then stop with that subset as sol.
4: If the subset is not feasible or if we have reached the end of the set then backtrack through the subset until we find the most suitable value.
5: If the subset is feasible then repeat step 2.
6: If we have visited all the elts without finding a suitable subset and if no backtracking is possible, the stop with no sol.

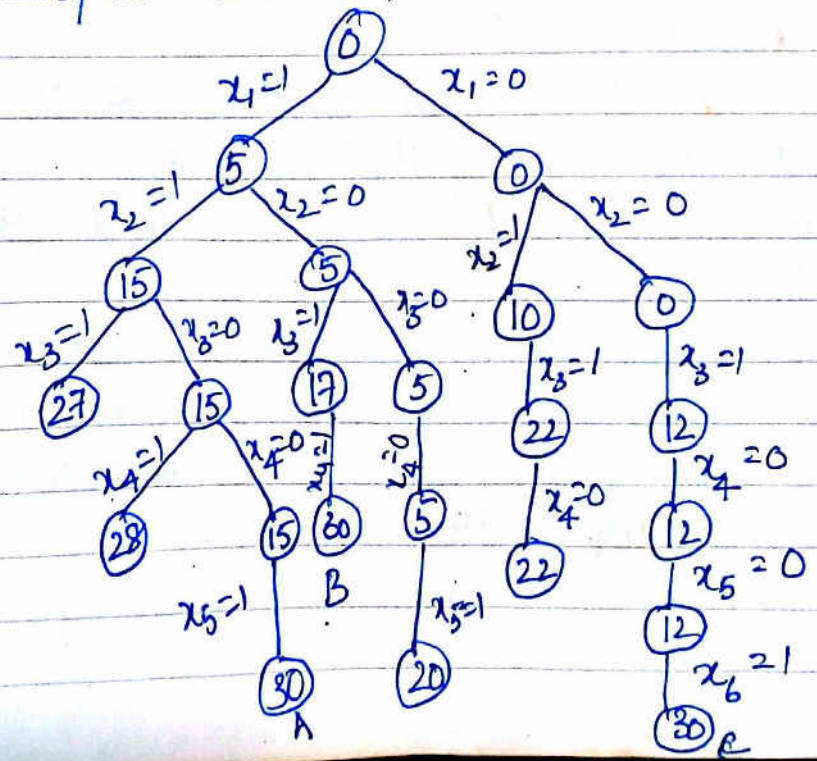
{6, 8, 12, 15, 18, 22}. N = 30

Eg: Consider a set S = {5, 10, 12, 13, 15, 18} and N = 30

Possible no of subsets considered for getting the required solution :

Subset {empty}	Sum 0	Action initially set is empty
5	5	
5, 10	15	
5, 10, 12	27	
5, 10, 12, 13	40	Sum exceeds N = 30 Hence backtrack
5, 10, 12, 15		Not feasible
5, 10, 12, 18		Not feasible
5, 10		List ends. Backtrack.
5, 10, 13	28	
5, 10, 13, 15	33	Not feasible. Backtrack
5, 10	15	
5, 10, 15	30	Solution obtained.

We can represent various sol to sum of subset by a state space tree as,



{7, 18, 19, 20, 8, 10, 12, 30} s = 10 ∴

Algorithm SS(sum, index, sum1)
// sum is a variable that stores the sum of all the selected elts, index denotes the index of chosen elt from the given set. sum1 is initially sum of all elts. And after selection of some elt from the set subtract the chosen value from sum1 each time.
- // w[1:n] represents the set containing n elts.
- // a[j] represents the subset where $1 < j \leq index$ and it is determined.
- // $sum = \sum_{j=1}^{index-1} w[j] * a[j]$.
- // $sum1 = \sum_{j=index}^{n} w[j]$, w[j] is sorted in non-decreasing order.
- // For a feasible sequence assume that $w[i] \leq m$ and // $\sum_{i=1}^{n} w[i] \geq m$.

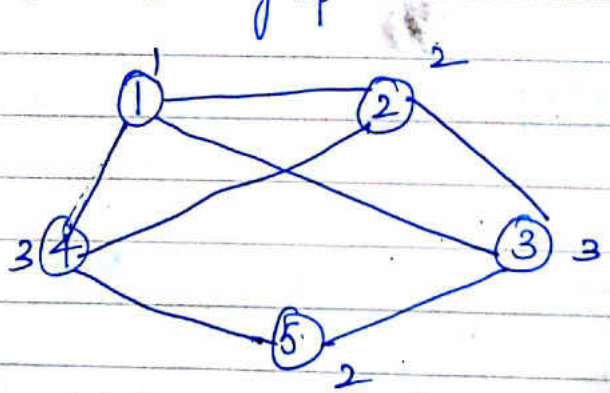
{ // Generate left child until sum + w[index] ≤ m.

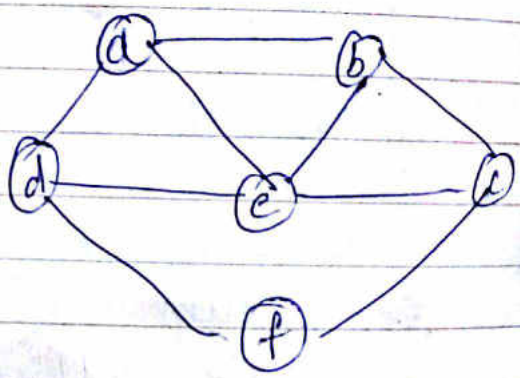
```
a[index] := 1;
if (sum + w[index] = m) then
  write (a[1 : index]);
else if (sum + w[index] + w[index + 1] ≤ m) then
Sum_Subset ((sum + w[index]), (index + 1),
                (sum1 - w[index]));
if (sum + sum1 - w[index] ≥ m) AND
   (sum + w[index + 1] ≤ m)) then
{
   a[index] := 0;
   Sum_Subset (sum, (index + 1), (sum1 - w[index]));
   // bounding fn is true
}
}
```

III. Graph Coloring Problem :-

Let G be a graph consisting of a set of vertices and a set of edges and let m be a given positive integer. Graph coloring is a problem of coloring each vertex in a graph in such a way that no two adjacent vertices have same color and yet m colors are used. This prob is also called as m - coloring prob. If the degree of given graph is d. Then we color it with d+1 colors. The min no of colors required to color the graph is called its chromatic number. The max chromatic no for any planar graph is 4.

Eg: Consider the graph:





Here the color of each node is indicated by a no just beside the node.

We require 3 colors to paint this graph. Hence the chromatic no of this graph is 3.

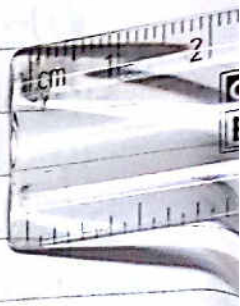
Using backtracking this problem can be solved as. Let the graph G consisting of n vertices with the adjacency matrix A.

ie., $A = [a_{ij}]_{n \times n}$, where $a_{ij} = 1$, if $(i,j) \in E(G)$
$= 0$, otherwise.

Let the colors are represented by the integers $(1, 2, \ldots m)$ and let $(x_1, x_2, \ldots x_n)$ be the solution, where x_i is the color of vertex i.

```
Algorithm m_coloring(k)
{
  repeat
    {
      NEXT VALUE(k);
      if (x[k] ← 0) then
      return;
      if (k = n) then
        write (x[1:n]);
      else m_coloring (k+1);
    }
    until (false);
}
Algorithm NEXTVALUE (k)
  // x[k] indicates the legal color assigned to the
  // vertex. if no color exists then x[k]=0.
{
    repeat
    {
      x[k] ← (x[k+1]) mod (m+1);
      if (x[k] = 0)) then
      return;
      for j ← 1 to n do
      {
```

```
        if ((A[k,j] ≠ 0) and (x[k] = x[j])) then
            break;
        }
        if (j = n+1) then return;
    } until(false);
}
```

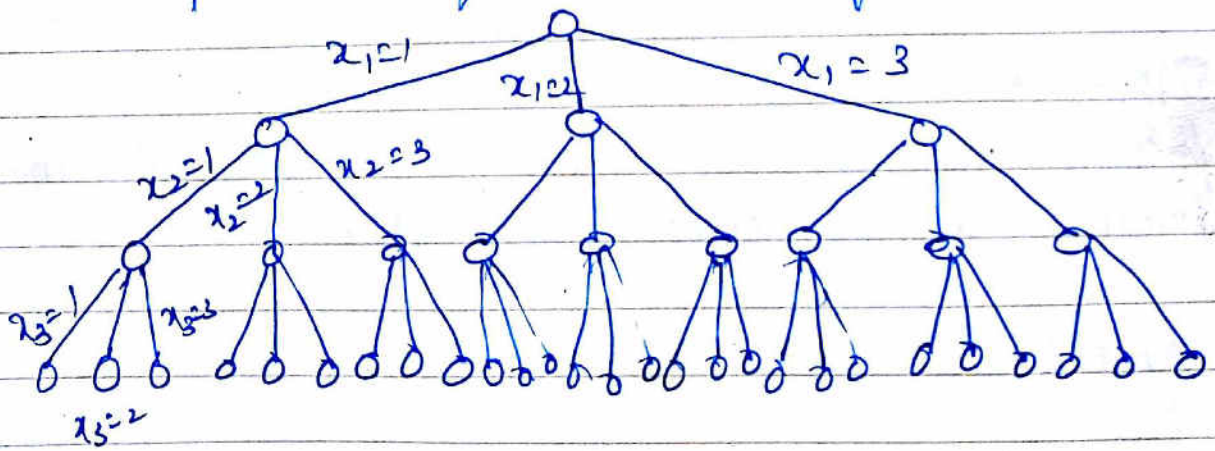


Time Complexity:

At each internal node $O(mn)$ time is spent by NEXTVALUE to determine the children corresponding to legal colorings. Hence, the total time is bounded by,

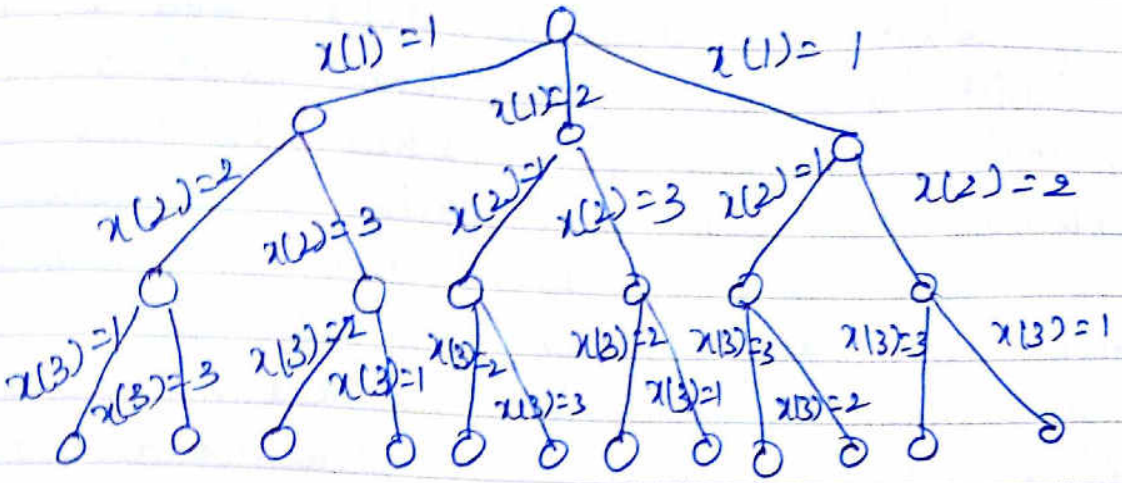
$$\sum_{i=1}^{n} m^i n = n(m + m^2 + \cdots + m^n)$$
$$= n \cdot m^n$$
$$= O(n m^n).$$

State Space Tree for Graph-coloring when $n=3$, $m=3$.



State space tree contains all possible moves where some moves leads to sol & some may not lead to sol.

The following tree contains only possible sol:



The possible sol are 12, within that 6 sol exist with exactly 2 colors. In the above tree, when $x(1)=1$, then $x(2)$ contain either 2 or 3 but not 1, since two adjustment nodes should not have same color, when $x(2)=2$, then $x(3)$ contain either 1 or 3. Similarly, we can process remaining nodes.

IV. Hamiltonian Cycles:-

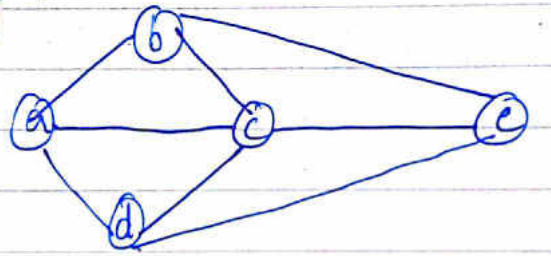
We know that, a path $x_0, x_1, \cdots x_{n-1}, x_n$ in the graph $G=(V,E)$ is called a Hamiltonian path, if $V=\{x_0, x_1, \cdots x_{n-1}, x_n\}$ and $x_i \neq x_j$ for $0 \leq i \leq j \leq n$.

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

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 next adjacent vertex is selected on the basis of alphabetical or numerical order.

If at any stage an arbitrary vertex, say 'x' make a cycle with any vertex other than vertex 'o' then we say that dead end is reached. In this case we backtrack one step and again the search begins by selecting another vertex. It should be noted that, after backtracking the edge from the partial sol must be removed. The search using backtracking is successful, if a Hamiltonian cycle is obtained.

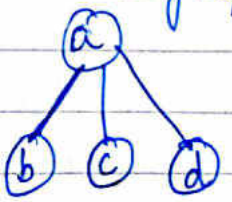
Eg: Consider a graph $G = (V, E)$ as shown below, we have to find a hamiltonian circuit using backtracking.



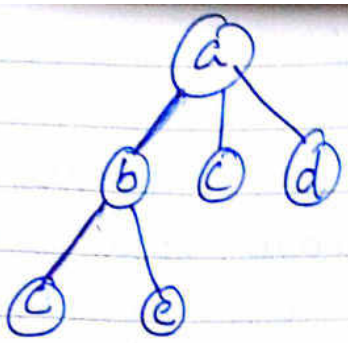
Initially, we start our search with vertex 'a', the vertex 'a' becomes the root of our implicit tree.

(a) ← root.

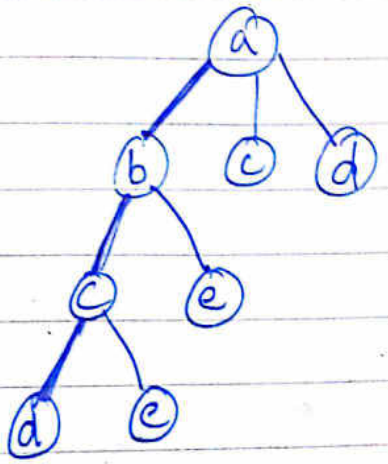
Next we choose vertex 'b' adjacent to 'a', as it comes first in lexicographical order (b, c, d).



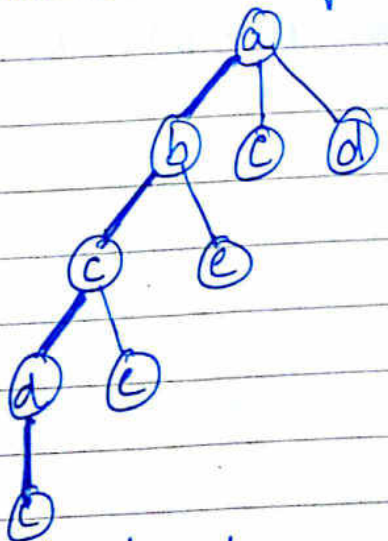
Next vertex 'c' is selected which is adjacent to 'b' and which comes first in order (c, e).



Next we select the vertex 'd' adjacent to 'c' which comes first in lexicographical order (d, e).



Next vertex 'e' is selected. If we choose vertex 'a' then we do not get the hamiltonian cycle.



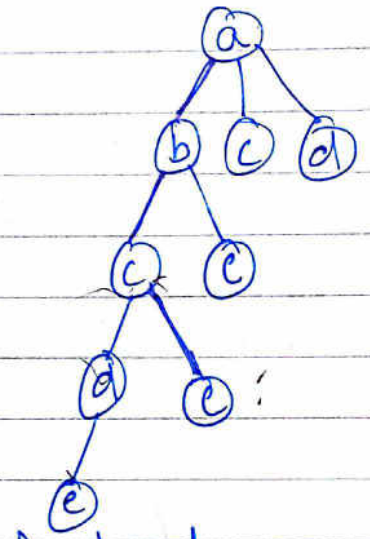
dead end.

The vertex adjacent to 'e' are b, c, d but they are already visited. Thus, we get the dead end. So, we backtrack one step and remove the vertex 'e'

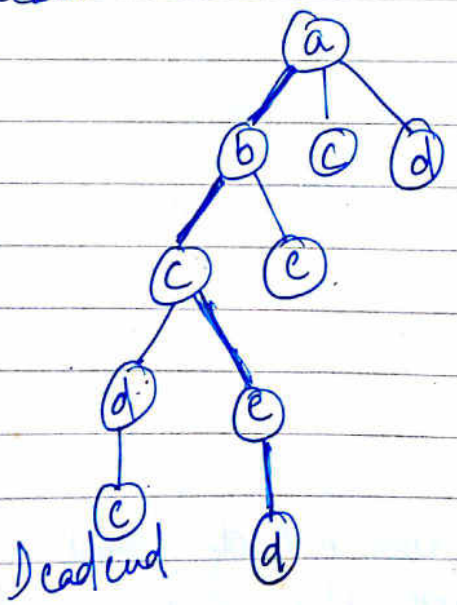
from ~~a~~ one partial sol.

The vertex adjacent to 'd' are e, c, a from which vertex 'e' has already been checked and we are left with vertex 'a' but by choosing vertex 'a' we do not get the hamiltonian cycle. So, we again backtrack one step.

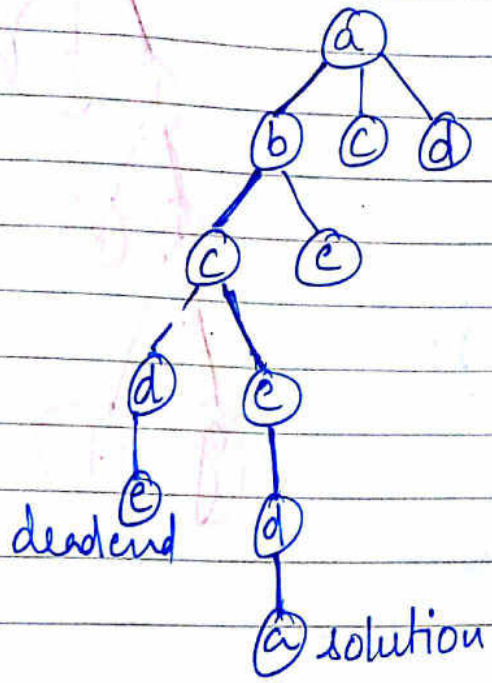
Hence, we select the vertex 'e' adjacent to 'c'.



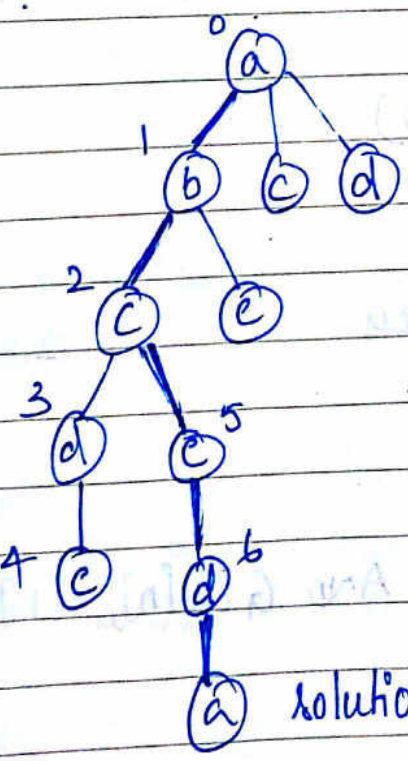
The vertex adjacent to 'e' are (b, c, d) so vertex d is deleted.

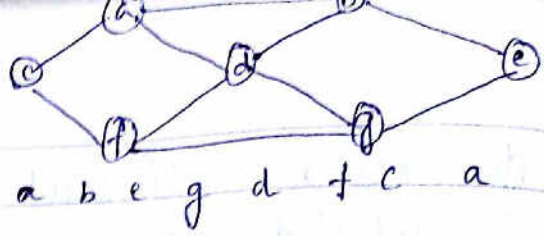


The vertex adjacent to 'd' are (a, c, e) so vertex 'a' is selected. Hence, we get the Hamiltonian cycle as all the vertex other than the start vertex 'a' is visited only once, a-b-c-e-d-a.

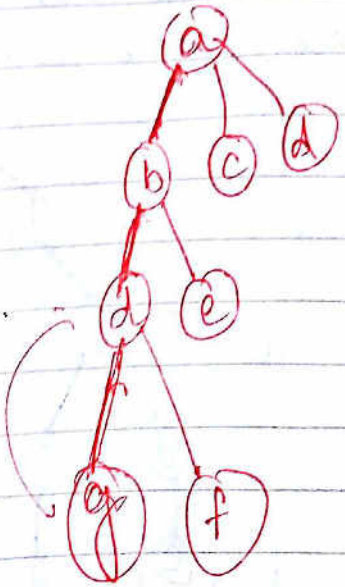


The final implicit tree for the Hamiltonian circuit is as shown below. The no above each node indicates the order in which they are visited.






```
Algorithm Hamiltonian (k)
{ repeat
  {
    Next vertex (k)
    if (x[k] = 0) then
     return;
     if (k = n) then
     write (x[1:n])
     else
     Hamiltonian (k+1);
  }
  until (false);
}
```




```
Algorithm Next Vertex (k)
{
  repeat
  {
   x[k] ← (x[k]+1) mod (n+1);
   if (x[k] = 0) then
   return;
   if ( G[x[k-1], x[k]] ≠ 0)
    {
     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);
}
```

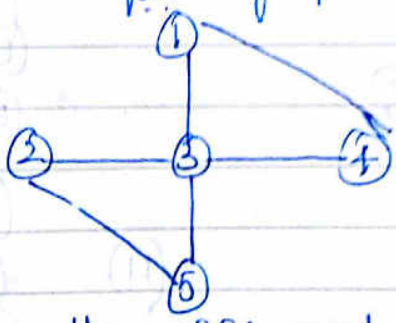
Spanning Trees and Connected Components:

A spanning tree for a connected, undirected graph $G = (V, E)$ is a subgroup of G that is an undirected tree and contains all the vertices of 'G'. We can also define a spanning tree as:

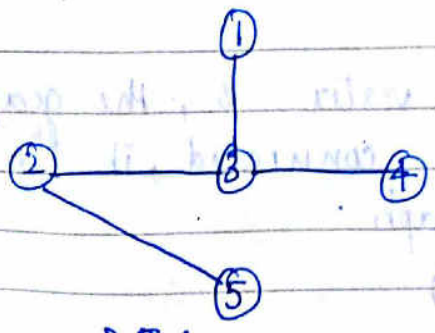
A subgraph $T = (V, E)$ of G is a spanning tree of G iff T is a tree. A spanning tree of a graph should include all the vertices and a subset of edges (E).

The BFS and the BFT algorithms are applied to test whether a graph G is connected or not and to obtain the connected components of G. Consider the set of all edges (u, w), where all vertices w are adjacent to u and are not visited.

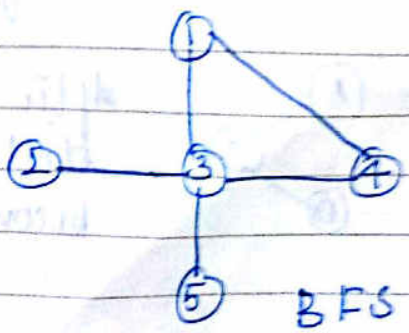
Consider the foll graph:



The foll are the DFS and BFS spanning trees for the above graph:



DFS



BFS.

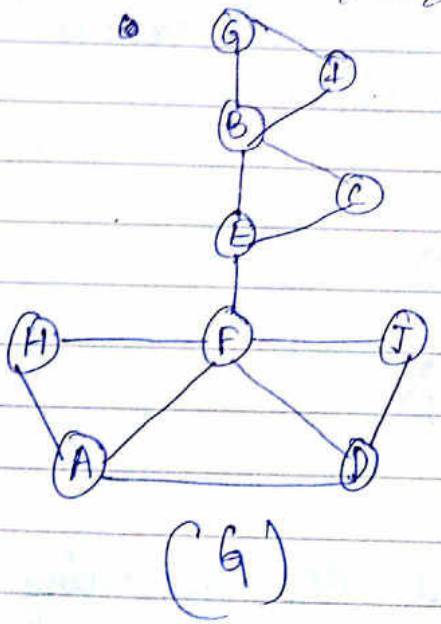
→ Biconnected Components:

A connected undirected graph G is said to be biconnected if it remains connected after removal of any one vertex and the edges that are incident upon that vertex.

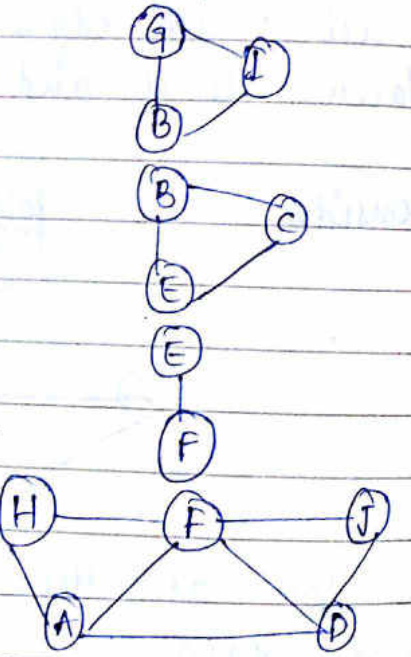
A biconnected component (bicomponent) of an undirected graph is a maximal biconnected subgraph, that is, a biconnected subgraph not contained in any larger biconnected subgraph.

→ Articulation Point (or Cut Point):-

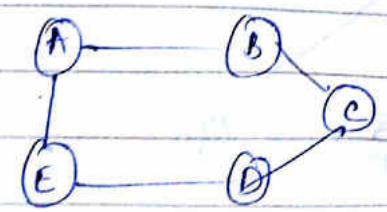
A vertex v is an articulation point



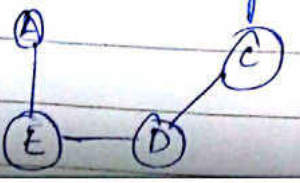
(G)



For eg:- Consider the following graph



After removing vertex B, the graph still remains connected, it is a biconnected graph.



Construction of Biconnected Graph:

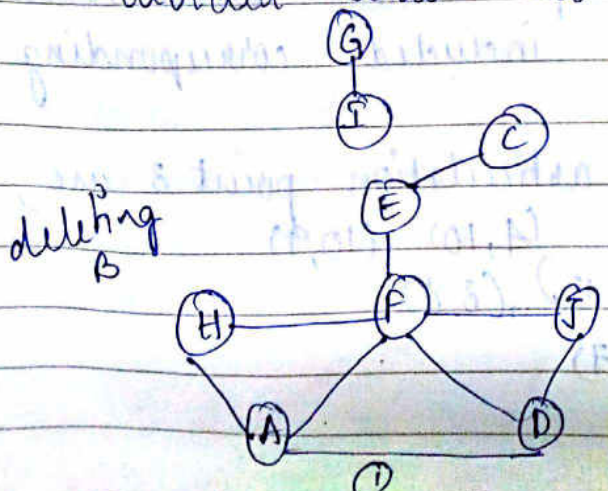
* Check the given graph whether it is biconnected or) not.
* If the given graph is not biconnected then identify all the articulation points.
* If articulation points exists, determine a set of edges whose inclusion makes the graph biconnected.

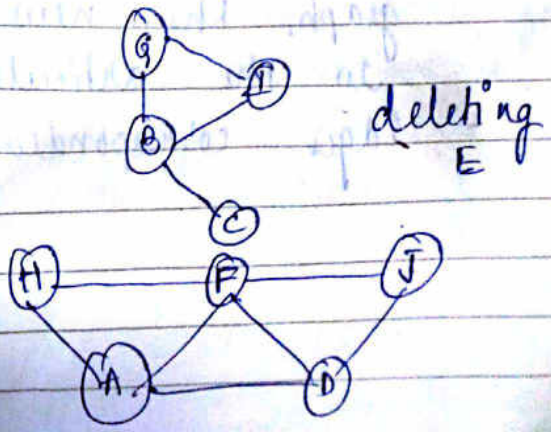
Articulation Point (or) Cut Point:

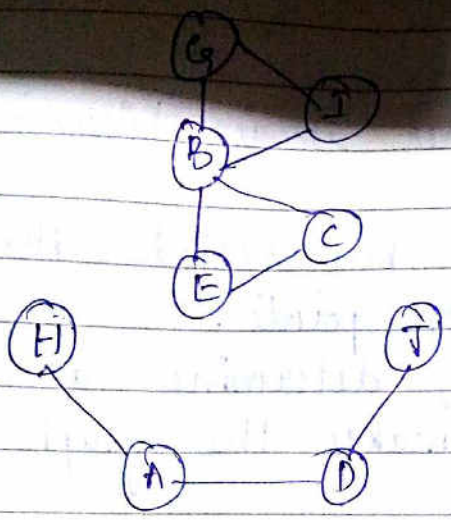
A vertex v is an articulation point or cut point for an undirected graph G, if there are distinct vertices u & x, distinct from v also such that v is in every path from u to x.

The definition of articulation pt or cut pt can be once again defined using simple terms (ie). A vertex v in a connected graph G is a cut point if and only if the deletion of a vertex v together with all edges incident to v disconnects the graph into two (or) more non-empty components.

Consider the graph (G), after deleting vertex B and incident edges of B, the given graph is divided into two nonempty components.





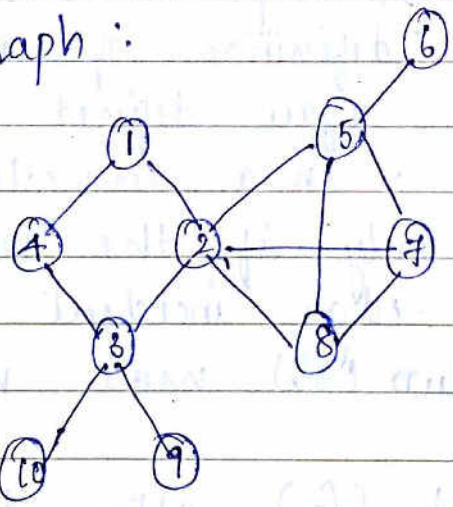


deleting F

→ Identifying the Articulation Points and Biconnected Components:-

If the given graph G has $n \geq 2$ vertices, the problem is efficiently solved using a depth first spanning tree of G.

Consider following graph:



- The articulation points are 2, 3, 5.
- To transform the above graph into biconnected graph, the new edges are included corresponding to the articulation point.
- Edges corresponding to the articulation point 3 are, (4,10), (10,9)

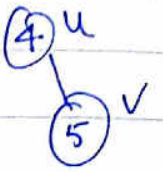
2 – (1,5), (3,8)

5 – (6,7)

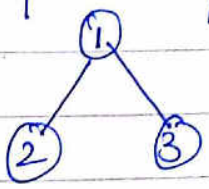
Depth First Spanning Tree Properties:-

These are very useful in identifying articulation points and biconnected components.

* If (u, v) is any tree edge of DFS spanning tree G, then either u is an ancestor of v or v is an ancestor of u.

eg 

* The root node of a depth first spanning tree is an articulation point iff it has atleast 2 children.



After deleting vertex 1 and incident edges of the same, two non empty components that exist is two children nodes (i.e 2, 3). Therefore vertex 1 is an articulation pt.

* → If u is any other vertex other than root, then it is not an articulation point iff from every child of w of u it is possible to reach an ancestor of u using only a path made up of descendents of w and a back edge.

Note that if this cannot be done for some child w of u then the deletion of vertex u leaves behind atleast two non empty components (one containing the root and other containing vertex w).

For each vertex, u, define L[u] as:

L[u] = min {