Scanned with CamScanner

Unit - III

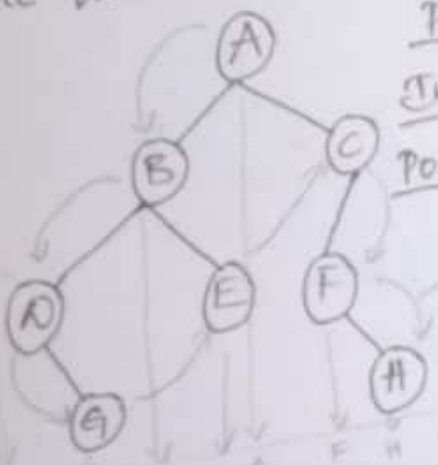
Techniques for binary trees:

Binary tree:
Binary tree is a tree which has atmost 2 child nodes

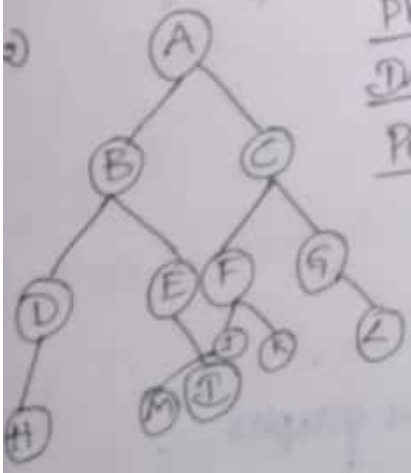
Traversal Techniques:
There are 3 types of traversal techniques in binary tree They are

1) Preorder : P, L, R (Parent, left, right)
2) Inorder : L, P, R (left, Parent, right)
3) Postorder : L, R, P (left, right, parent)

Find the traversal techniques for the following tree



Preorder: ABDGECFH
Inorder: DGBEAFHC
Postorder: GDEBHFCA



Preorder: ABDHEICFJMKGL
Inorder: HDBEIAMJFKCGL
Postorder: HDIEBMJKFLGCA

Algorithm:

```
treenode = record;
{
type data;
treenode *lchild;
```

Scanned with CamScanner

Scanned with CamScanner

Scanned with CamScanner

```
treenode *rchild;
};
Algorithm Inorder(t)
{
if t≠0 then   → tree is not empty
 {
Inorder(t->lchild);
 Visit(t);
Inorder(t->rchild);
 }
}
Algorithm Preorder(t)
{
 if t≠0 then
{
  Visit(t);
  preorder(t->lchild);
  preorder(t->rchild);
   }
}
Algorithm postorder(t)
{
 if t≠0 then
{ postorder(t->lchild);
  postorder(t->rchild);
   Visit(t)
   }
}
```

Techniques for graphs:-

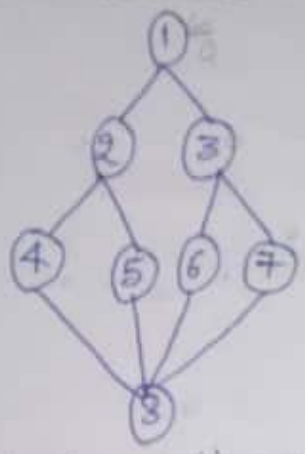
There are two traversal techniques in graphs

1) Breadth First search
2) Depth First search

Scanned with CamScanner

Scanned with CamScanner

1) Breadth First Search:



Adjacency list for the graph:

Algorithm:

```
Algorithm BFS(v)
{
  u:=v;
  visited[u]:=1;
repeat
{
for all vertices w adjacent from u do
  {
if (visited[w]=0) then
     visited[w]:=1;
  }
}
if queue is empty then
     return
  else
delete the next element u from queue;
```

98/107

Scanned with CamScanner

```
} until (false);
}
Algorithm BFS (G,n)
{
for i:=1 to n do
    visited[i]:=0;
for i:=1 to n do
if (visited[i]=0) then
    BFS (G,i);
}
```

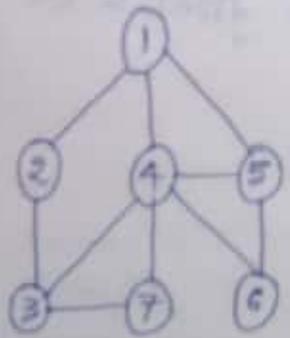
Connected components:-

The Connected Components of a graph is a collection of vertices such that there is a path b/n every pair of vertices in the same component that means there exists a path b/n V_i and V_j, If V_i and V_j are in the same component. A graph is said to be connected if there exists a path b/n any two vertices.

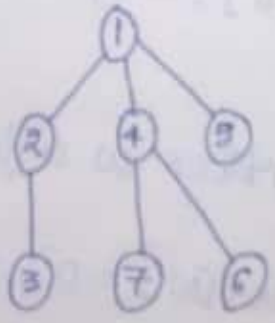
If the graph G is connected undirected graph then we can visit all the vertices of the graph in first call to BFS or DFS.

The subgraph which we obtain after traversing the graph using BFS or DFS represents the connected component of the graph.

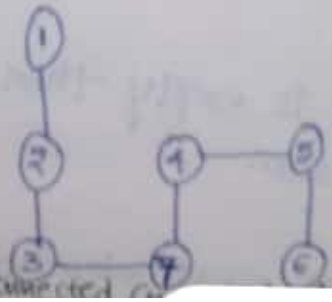
Example:



'G'



Connected Component by BF



Connected C... DFS

Scanned with CamScanner

Scanned with CamScanner

Algorithm:

```
Algorithm ConnComponents (G,n)
{
 for (i←1 to n) do          → O(n+E)
  {
    visited[i] ← 0;
  }
for (i←1 to n) do
 {
 if (visited[i]=0) then
     DFS (i);
Output the newly visited vertices with adjacent edges
  }
}
```

Time Complexity:

The time complexity of the above algorithm is O(n+E). In that, the total time taken by DFS is O(E) and the for loop which DFS is called takes O(n) time.

Strongly Connected Components:

Finding strongly connected components of a directed graph is one of the application of DFS. In undirected graph, two vertices are connected if they have path connecting them. But in case of directed graph vertex A is strongly connected to B if there exists path from A to B & B to A. The relation b/w the vertices should satisfy the following properties:

i) Reflexive:

Any vertex is strongly connected to itself

ii) Symmetric:

If any vertex 'u' is connected to vertex 'v', then vertex v is connected to u then those two vertices satisfy the Symmetric property

Scanned with CamScanner

Scanned with CamScanner

iii) transitive:
For a strongly connected component if vertex 'u' is connected to vertex 'v', v is connected to 'w' with the Path u-v, v-u, v-w, w-v. This holds the transiti property u=v=w.

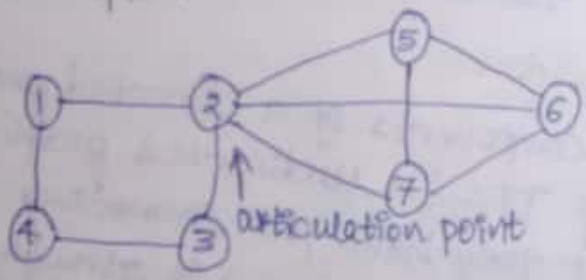
Biconnected Components and DFS:-

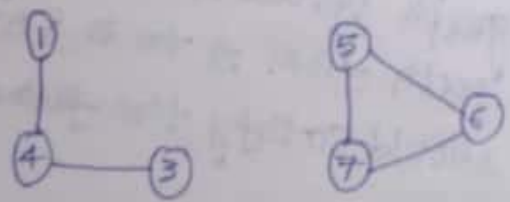
In this section, we study two concepts
1) Articulation point and the other is
2) Biconnected Components

DFS technique is used to find articulation point and biconnected components

Definition of articulation point:
Let G=(V,E') be a connected undirected graph then an articulation point of graph 'G' is a vertex whose removal disjoints the graph 'G'. The articulation point is also called Cut vertex.

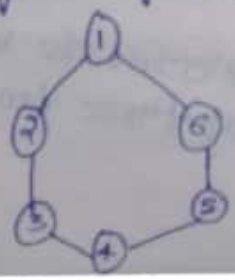
Example:





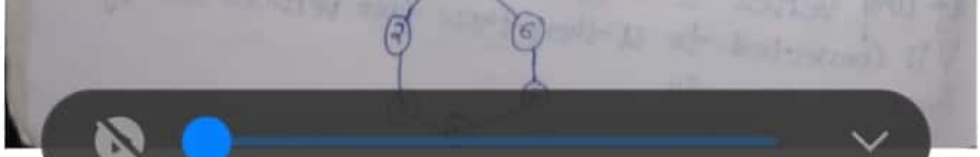
A graph 'G' is said to be biconnected if it contains no articulation point. That means if we remove any single vertex we do not get any disjoint graphs

Example:



Scanned with CamScanner

Scanned with CamScanner



Identification of articulation Points:

i) The easiest method to remove a vertex and its corresponding edges one by one from graph 'G' and test whether the resulting graph is disconnected or not. The time complexity for this is O(V+E)

ii) Another method is using the DFS technique to find the articulation points

Procedure:

First we need to apply the DFS technique to the graph 'G'. After performing DFS onto the given graph we get a DFS tree

2) While building the DFS tree, number is alloted for each vertex. These numbers indicate the order in which a DFS visits the vertices. These numbers are called Depth First Search Numbers (DFN)

3) While building the DFS tree we can classify the edges into 4 categories

i) tree edge - it is an edge in DFS tree

ii) Back edge - it is an edge u,v which is not in DFS-tree and v is ancestor of u. It indicates a loop

iii) Forward edge - An edge u,v which is not in tree and u is an ancestor of v

iv) Cross edge - An edge u,v is not in DFS tree and v is neither an ancestor nor a descendant of u

To identify the articulation point following observations can be made

i) The root of the DFS tree is an articulation point if it has two (or) more children.

ii) A leaf node of DFS tree is not an articulation tree

iii) if u is an internal node then it is not an articulation

point if and only if $L(w) \geq DFN(u)$ is possible to reach an ancestor of 'u' using a path made up of descendence of w and back edge.

The equation to find the articulation point is as follows

Scanned with CamScanner

Scanned with CamScanner

point if and only if it is possible to reach an ancestor of 'u' using a path made up of descendence of w and back edge.

The equation to find the articulation point is as follows

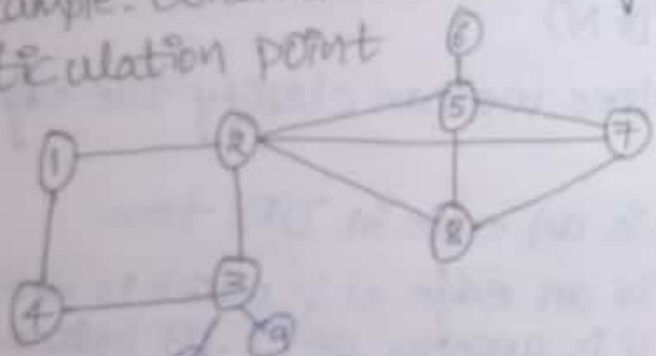
$$low[u] = \min\{dfn[u], \min\{low[w] / w \text{ is a child of } u\}, \min\{dfn[w] / (u,w) \text{ is a back edge}\}\}$$

where low[u] is the lowest depth-first number that can be reached from 'u' using a path of descendence followed by atmost one package.

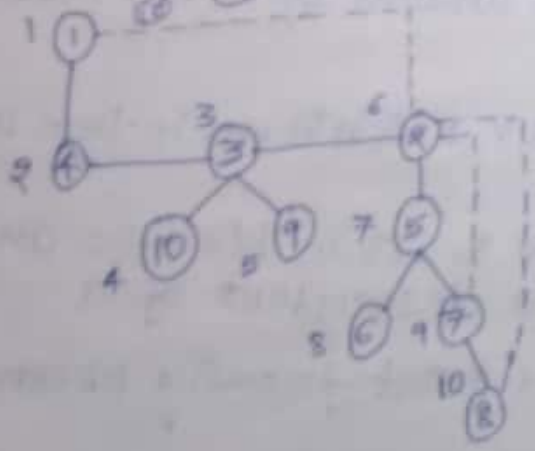
The vertex 'u' is an articulation point when the following condition is satisfied

$$low[w] \geq dfn[u]$$

Example: Consider the below graph and identify the articulation point



Sol:



Let us compute low[u] using the formula

$$low[u] = \min\{dfn[u], \min\{low[w] / w \text{ is a child of } u\}, \min\{dfn[w] / (u,w) \text{ is a back edge}\}\}$$

Scanned with CamScanner

Low[1] = min{dfn[1], min{Low[4]}, min{dfn[7]}}
= min{1, Low[4], 6}
= 1

Low[9] = min{dfn[2], min{Low[5]}, min{dfn[1], dfn[7], dfn[
= min{6, Low[5], min{1, 9, 10}}
= min{6, Low[5], 1}
= 1

Low[3] = min{dfn[3], min{Low[10], Low[9], Low[7]}, no back edge}
= min{3, min{Low[10], Low[9], 1}}
= 1

Low[4] = min{dfn[4], min{Low[3]}, No backedge}
= min{2, 1}
= 1

Low[5] = min{dfn[5], min{Low[7]}, min{dfn[7]}}
= min{7, min{Low[7], 6}}
= min{dfn[5], min{Low[6], Low[7]}, no back edge}
= min{5, ...}

Low[6] = min{dfn[6], min no child, no back edge}
=

Low[5] will not be decided until calculating the Low[6] & Low[7]
∴ Low[5] keep it as it is.

Now Low[6] = min{dfn[6]}
= 8

Low[7] = min{dfn[7], min{Low[8]}, min{dfn[2]}}
= min{9, Low[8], 6} = min{9, 6, 6} = 6

Low[8] = min{dfn[3], —, min{dfn[2]}}
= min{10, 6}
= 6

Low[5] = min{7, min{8, 6}}
= min{7, 6}
= 6

Scanned with CamScanner

low[9] = min{dfn[9]} = 5
low[10] = min{dfn[10]} = 4

The low values are low[1:10] = {1,1,1,1,6,8,6,6,5,4}

Now by checking the condition low[w] ≥ dfn[u]. We need to identify the articulation point → {1,2,3,5}

Vertex '2' is an articulation point because child of '2' is '5' low[5] = 6 and dfn[2] = 6

Similarly vertex '3' is also an articulation point because children of '3' is 10 and 9

low[10] = 4
low[9] = 5
dfn[3] = 3

Vertex '5' is also an articulation point, the children of '5' are 6 & 7 low[6] = 8 and low[7] = 6, dfn[5] = 7

∴ 2,3 & 5 are the articulation points

Algorithm:

```
Algorithm DFS_Art(u,v)
{
 dfn[u] := dfn_num;
 low[u] := dfn_num;
 def_num := dfn_num+1;
 for (each vertex w adjacent to u) do
  {
  if (dfn[w]=0) then
    {
     DFS_Art (w,u);
   low[u] := min {low[u], low[w]};
    }
 else
  if (w' = u ) then
 low[u] := min{low[u], dfn[w]};
    }
 }
```


Scanned with CamScanner

Scanned with CamScanner

Identification of Biconnected Components:

A Biconnected graph $G=(V,E)$ is a connected graph which don't have articulation points. A biconnected component of a Graph G is a maximal biconnected subgraph that means it is not contained in any larger biconnected subgraph of G.

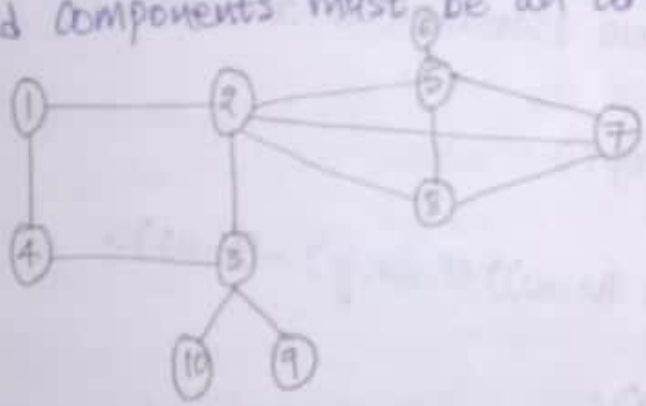
Some key observations can be made in regard to biconnected components of graph are

1) Two different biconnected components should not have any common edges

2) Two different biconnected components can have common vertex.

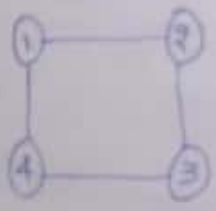
3) A common vertex which is attaching two(or) more biconnected components must be an articulation point

Example:

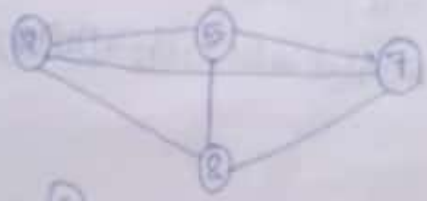


Here the articulation points are 2, 3 & 5. Based on these points the biconnected components are:

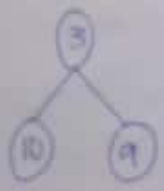
1)



2)



3)



4)

Scanned with CamScanner

```
Low[v] = m Biconnect(u,v)
{
dfn[u] ← dfn_num;
low[u] ← dfn_num;
dfn_num ← dfn_num+1;
for (each vertex w adjacent to u) do
{
if ((v!=w) and (dfn[w] < dfn[u])) then
   push (u,w) on to the stack st;
if (dfn[w] = 0) then
{
if (low[w] ≥ dfn[u]) then
{
write("obtained Articulation points");
write("The new biconnected component");
repeat
{
edge (x,y) ← pop()
write (x,y);
} until ((x,y) = (u,w)) OR ((x,y) = (w,u));
}
Biconnect (w,u);
low[u] ← min (low[u], low[w]);
}
else if (w!=u) then
low[u] ← min(low[u], dfn[w]);
}
}
```


Scanned with CamScanner

Backtracking:-

General method:-

In the backtracking method, the desired solution is expressable as an n-tuple $(x_1, x_2, x_3, \ldots x_n)$ where x_i is choosen from finite set S_i. The solution maximises, minimises or satisfies a criterian function $C(x_1, x_2, \ldots x_n)$. The problem in the backtracking can be categorised into 3 categories:

1) Decision problem - find whether there is an feasible solution
2) optimisation problem - find the best solution
3) Enumeration problem - find the feasible solutions among all the feasible solutions

In this method for problem 'P', C be the constraint and we need to get the solution by satisfying 'C'.

The basic idea of Backtracking is to build up a vector, one component at a time and to test whether the vector been formed any chances of success. The major advantage of this method is that we can realize the fact that the partial vector does not lead to an optimal solution. In such a situation, that vector can be ignored. Backtracking is a Depth First Search with some bounding functions. All solutions using Backtracking should satisfy a complex set of constraints, the constraints may be explicit (or) Implicit.

→ Explicit constraints are rules which determine the tuples in the solution space actually satisfy the criterion function

→ Explicit Constraints are the rules which restrict each vector element to be choosen from the given set

Scanned with CamScanner

Applications of Backtracking:

1) 8-Queens Problem
2) Sum of subsets problem
3) Graph coloring
4) Hamiltonian cycle
5) knapsack problem

Some terminologies used in BackTracking:

BT Algorithms determine the problem solutions by systematically searching the solution by tree structure

1) Each node in a tree is called a problem state
2) All paths from the root to other nodes define the State Space of the problem
3) The solution states are those problem states for which the path from root to problem state 's' defines a tuple in the solution states
4) In some trees the leaves define the solution states

Answer States:

These are the leaf nodes which corresponds to an element in the set of solutions. These are the solutions which satisfy the implicit constraint

Live Node:

A node which is been generated and all whose children have not yet been generated is called live node.

E-node: The live node whose children are currently being expanded is called E-node

Dead node: It is generated where the tree is not expanded further (or) all the children have been generated

Scanned with CamScanner

Scanned with CamScanner

Control abstraction: (algorithm for backtracking using recursive technique)

```
Algorithm Backtracking()
{
for(each a_k that belongs to T[a_1, a_2, ---a_{R-1}]) do
{
if (B_R(a_1, a_2, --- a_K) = true) then
{
if((a_1, a_2, ---a_R) is a path to answer node) then
print (a[1], a[2], ---a[K]);
    if (k<n) then
     backtrack(K+1);
      }
    }
}
```

Algorithm for Backtracking using Non-recursive technique

```
Algorithm Non-RecBacktracking(n)
    {
    k := 1;
while(k ≠ 0) then
 {
if((any a[k] that belongs to T(a[1], a[2], --a[k])
   remains untried) AND (B_k(a[1], a[2]--a[k]=true))
  then
{
if ((a[1], a[2]--a[k]) is a path to answer node) then
  write(a[1], a[2], ---a[k]);
  if (k<n) then
    {
    return k+1;
     }
   }
}
```

Scanned with CamScanner

Scanned with CamScanner

8-Queens Problem:-

This is a problem based on chess games. With this problem it is stated that arrange the 8 queens on the chess board in such a way that no two queens can attack each other. Here attack means no two queens can be placed in the same row, same column or on the same diagonal. When we place the queens in the chess board we need to check the diagonal conflicts by using the following formula

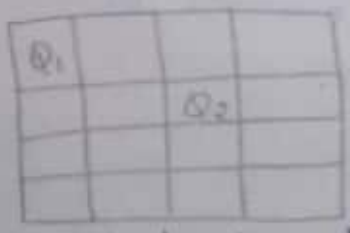
$$i+1 = k+1 \text{ (or) } i-1 = k-1$$

The above condition can also be checked as $|i-j| = |k-l|$

Example: First consider 4X4 chess board to place 4 queens.

Solution: Now we start with empty chess board, place Queen 1 on the first possible location i.e on the first row and first column.

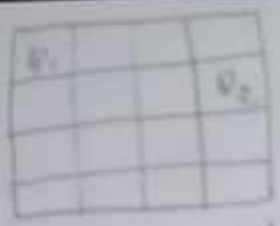
Step 2: Then place Q_2 after trying some possible locations at (2,3)



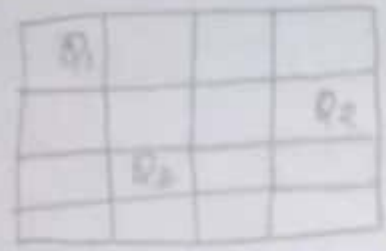
This is the end dead end, because third queen cannot be placed in the next column as there is no acceptable position for Q_3. Hence apply the BT algorithm and place the second queen at (2,4) location

Scanned with CamScanner

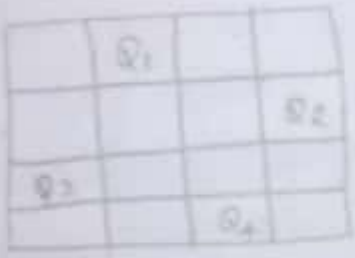
Scanned with CamScanner



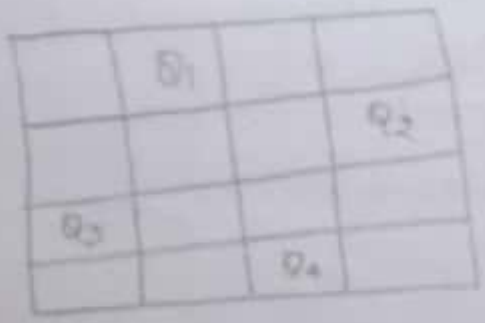
Now place the 3rd queen at (3,2) position but it is again another dead end as next queen '4' cannot be placed at permissible locations.



Hence we need to backtrack all the way to Q_1 and move it to (1,2) and place 'Q_2' at (2,4), queen '3' at (3,1) and queen '4' at (4,3)



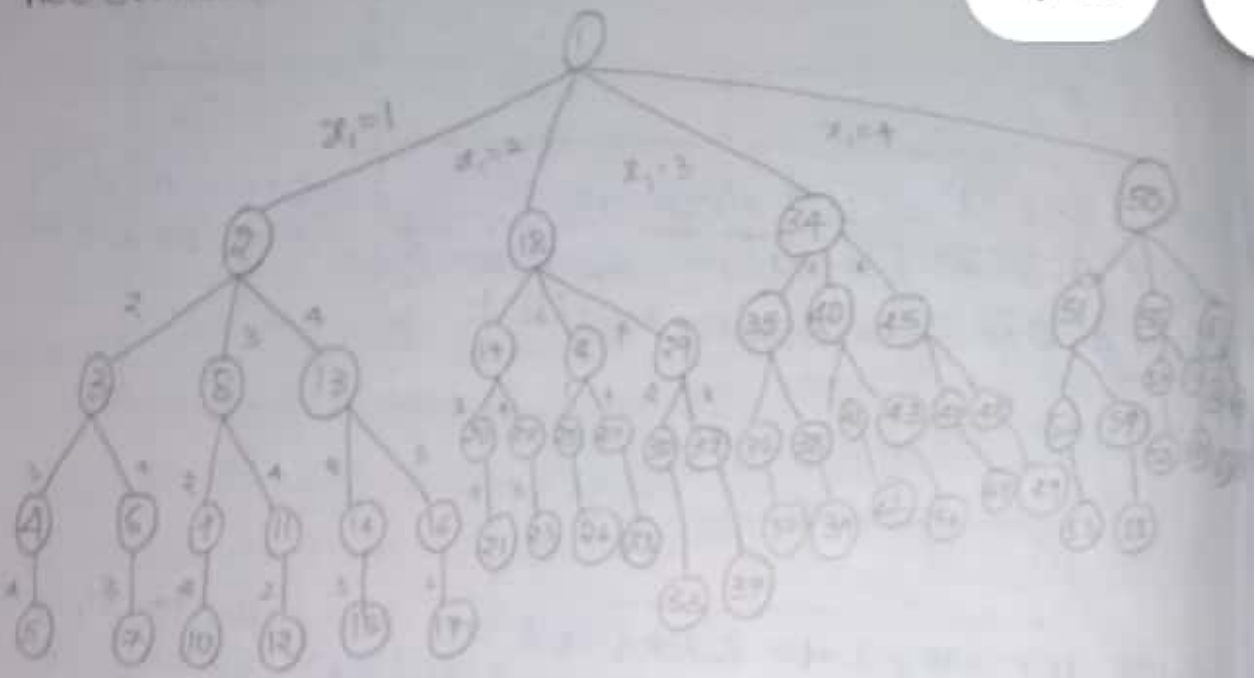
Therefore the solution for 4-queens problem is as follows:



Scanned with CamScanner

Scanned with CamScanner

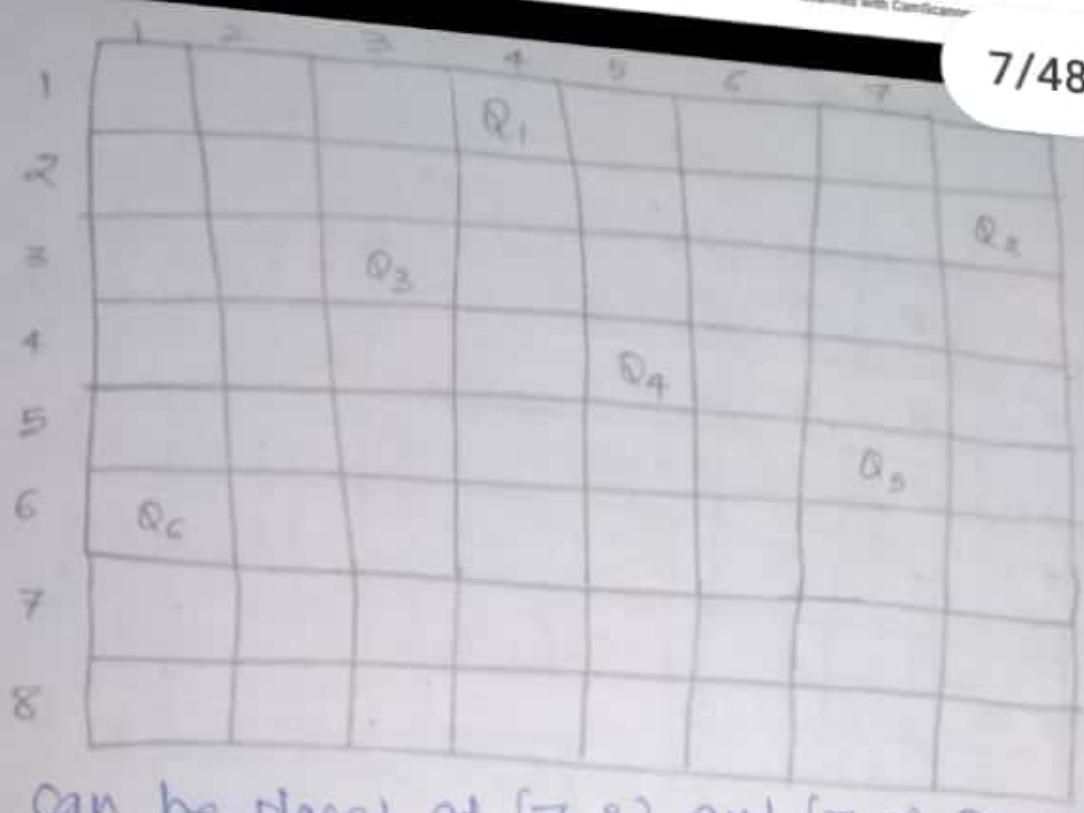
Tree structure:



Now we will consider how to place 8 queens on 8*8 chessboard

Thus we have placed '5' queens in such a way that no two queens can attack each other. Now we need to place Q_6 at location (6,6) then Q_5 can attack. If Q_6 is placed at (6,7) then Q_1 can attack; if Q_6 is placed at (6,8) then Q_2 can attack.
Therefore the Q_6 we need to backtrack and change the previously placed queens position.

Scanned with CamScanner



Q_7 can be placed at (7,2) and (7,8). If we place Q_7 at (7,2) then Q_6 can attack. If we place at (7,8) then Q_4 may attack Q_7. Hence we have to backtrack to adjust already placed queens

But Q_8 cannot be placed at location (8,5). Hence we need to backtrack finally the successful placement of 8-Queens is given below

Scanned with CamScanner

Scanned with CamScanner

Scanned with CamScanner


```
Algorithm place(k,i)
{
for j:=1 to k-1 do
if (x[j]=i) or (Abs(x[j]-i)=Abs(j-k)) then
    return false;
  return true;
}
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
```

Scanned with CamScanner

Scanned with CamScanner


```
Nqueens(k,n);
{
}
}
```

1) Solve 8-queens problem with a feasible solution Sequence (6,4,7,1)

sol:

The diagonal conflicts can be checked by the formula

$$i-j = k-l \longrightarrow (1)$$

(or)

$$i+j = k+l \longrightarrow (2)$$

The eq(1) implies $j-l = i-k$

eq(2) $\Rightarrow j-l = k-i$

$\therefore$ 2 queens lie on the same diagonal if & only if

$|j-l| = |i-k|$

Let $P_1=(i,j)$ and $P_2=(k,l)$ are two positions the P_1 and P_2 are the positions that are on the same diagonal if $i-k = j-l$ $i+j = k+l$ (or) $i-j = k-l$

Scanned with CamScanner

Scanned with CamScanner

Row \ Queens positions	1	2	3	4	5	6	7	8	Actions
1	6	4	7	1					
2	6	4	7	1	2				$i+j=k+l$ or $i-j=k-l$ $4+1=5+2$ or $4-1=5-2$ $5=7$ $3=3$
3	6	4	7	1	3				$4+1=6+3$ NO Conflict $4-1=5-3$ B_3 at 3rd column
4	6	4	7	1	3	2			$5+3=6+2$ Conflict $5-3=6-2$
5	6	4	7	1	3	5			$5+3=6+5$ NO Conflict $5-3=6-5$ Q_5 placed at 6th column
6	6	4	7	1	3	5	2		$6+5=6-5$ $6-5=7-2$ No Conflict
7	6	4	7	1	3	5	8	8	$7+8=8+8$ $7-8=8-8$ NO Conflict
8	6	4							

Graph Colouring:

Graph colouring is a problem of colouring each vertex in a graph in such a way that no two adjacent vertices have same colour and M colours are used to colour the vertices. This problem is also called as m colouring problem.

Scanned with CamScanner

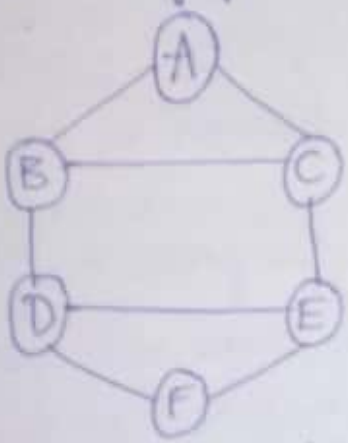
Scanned with CamScanner

If the degree of given graph is 'd' then we can colour it with 'd+1' colours. The least number of colours required to colour the graph is called chromatic number.

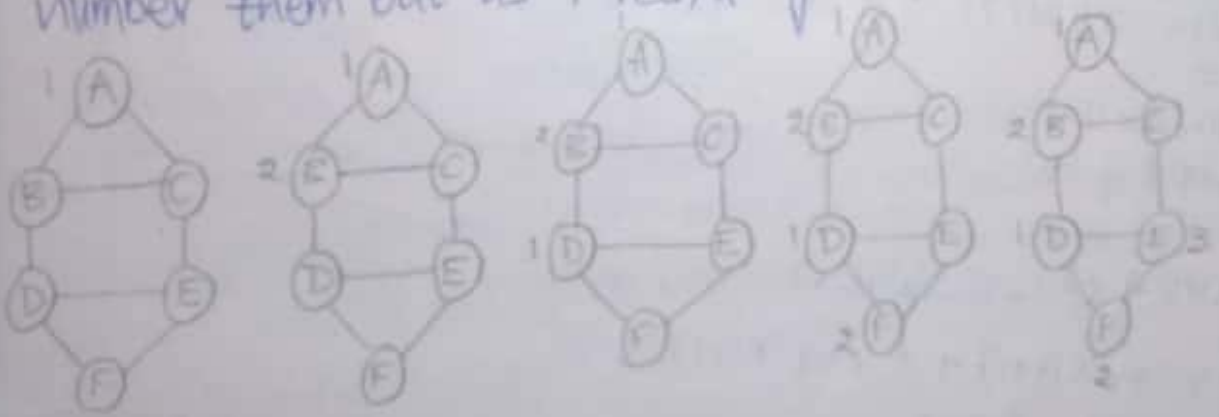
Eg: Hence we require 3 colours to colour the graph then the chromatic number of given graph is 3. We use backtracking technique to solve this problem.

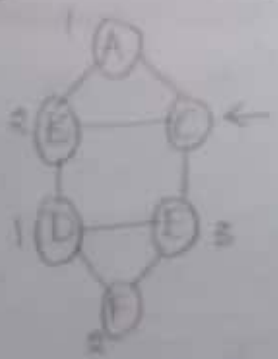
Consider the following graph

11/48



1) Graph G consists of vertices from A to F
2) There are 3 colours used red, green and blue. We will number them out as 1-red, 2-green, 3-blue

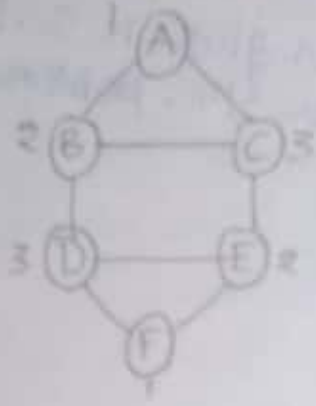




We cannot assign the colour either 1 or 2 or 3 for 'c' because the graph colouring problem constraint is that no two adjacent vertices have the same colour

Scanned with CamScanner

If we assign colour 1 to C then adjacent Vertex of C is A is having colour 1. Similarly B is having 2 and E is having 3.
Therefore we need to backtracking.



12/48

Algorithm:

```
Algorithm Gr.Color(k)
{
 repeat
{
  Gen-Col-Value(k);
 if (x[k]=0) then
 return;
if (k=n) then
write (x[1:n]);
else
 Gr-Color(k+1);
} until (false);
}
Algorithm Gen_Col_value(k)
{
 x(k) ← (x[k+1] + 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);
```

Scanned with CamScanner

Scanned with CamScanner

The graph Color takes computing time of $\sum_{i=0}^{n-1} m^i$
Hence total computing time for this algorithm is $O(nm^n)$

Sum of Subsets problem:

Let $S=\{s_1, s_2, \ldots s_n\}$ be a set of n +ve 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 ascending order i.e., $s_1 \leq s_2 \leq \ldots \leq s_n$

General Algorithm:
Let s be a set of elements and d is expected sum of subsets then

1) Starts with an empty set
2) Add to subset the next element from the list
3) If the subset is having the sum = d then stop with that subset as solution
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 elements without finding a suitable subset and if no backtracking is possible then stop the algorithm without solution.

Eg: Consider set $S=\{5, 10, 12, 13, 15, 18\}$ and $d=30$
solve it for obtaining sum of subsets

13/48

Scanned with CamScanner

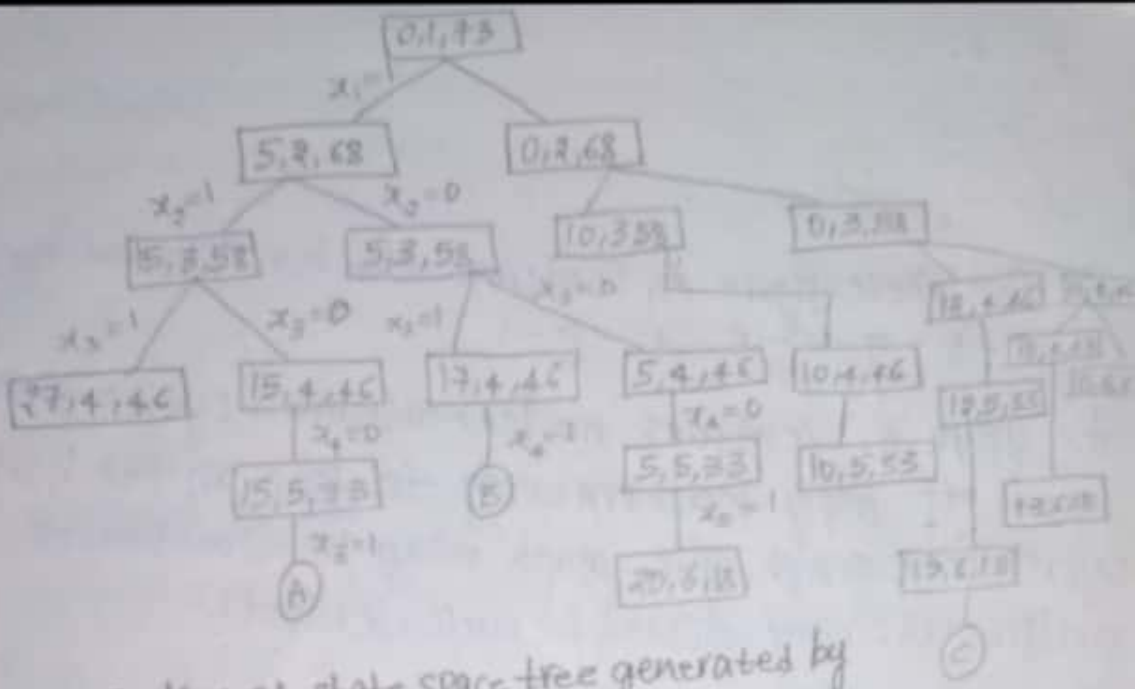
Sol: Initially Subset = { }	Sum = 0	0 ≤ 30 ✓ Then add next elem
5	Sum = 5	5 ≤ 30 ✓ Then add next elem
5, 10	Sum = 15	15 ≤ 30 ✓ Then add next elem
5, 10, 12	Sum = 27	27 ≤ 30 ✓ Then add next ele
5, 10, 12, 13	Sum = 40	40 ≤ 30 ✗ Perform backtrack
5, 10, 12, 15	Sum = 42	42 ≤ 30 ✗ Perform backtrack
5, 10, 12, 18	Sum = 45	45 ≤ 30 ✗ Perform backtrack as sum exceeds 'd' value
5, 10, 13	Sum = 28	28 ≤ 30 ✓ Then add next elem
5, 10, 13, 15	Sum = 43	43 ≤ 30 ✗ Perform backtrack
5, 10, 13, 18	Sum = 46	46 ≤ 30 ✗ Perform backtrack
5, 10, 15	Sum = 30	30 ≤ 30

∴ Subset = {5, 10, 15}

5, 10, 12, 13, 15, 18

Scanned with CamScanner

Scanned with CamScanner



Portion of state space tree generated by SumOfSub

15/48

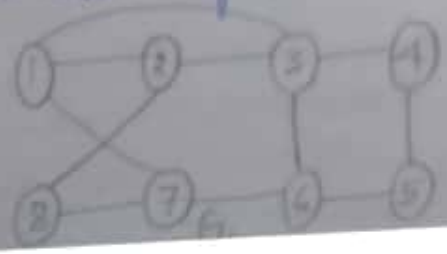
Algorithm:

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

Hamiltonian Cycle:-

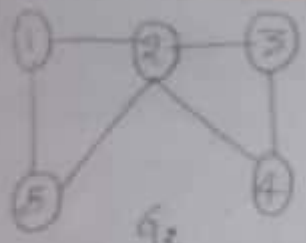
Let $G=(V,E)$ be a connected graph with n vertices. A Hamiltonian cycle is a round trip path along 'n' edges of graph G that visits every vertex once and returns to its starting position

Eg:



Scanned with CamScanner

Scanned with CamScanner



G_2

In the above figure 'G_1' contains the hamiltonian cycle
1-2-8-7-6-5-4-3-1

The graph 'G_2' contains no hamiltonian cycles. Here we need to apply backtracking to find all the hamiltonian cycles in a graph. The graph may be directed or undirected. only distinct cycles/outputs.

The backtracking solution vector $(x_1, \ldots x_n)$ is defined so that x_i represents the i^{th} visited vertex of proposed cycle.

Algorithm: To calculate next vertex

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

16/48

Scanned with CamScanner

Scanned with CamScanner

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


0/1 knapsack problem:-

Given n positive weights w_i, n positive profits p_i and a positive number m that is the knapsack capacity, the problem calls for choosing a subset of weights such that:

$\sum_{1 \le i \le n} w_i x_i \le m$ and $\sum_{1 \le i \le n} p_i x_i$ is maximized

The x_i's constitute a zero-one-valued vector. The solution space for this problem consists of 2^n distinct ways to assign zero or one values to the x_i's

Bounding Functions are needed to kill some live nodes without expanding them. A good bounding function for this problem is obtained by using an upperbound on the value of the best feasible solution obtainable by expanding the given live node and any of its descendants. If this upperbound is not higher than the value of the best solution determined so far, then that live node can be killed.

We continue the discussion using the fixed tuple size formulation. If at node z the values of $x_i, 1 \le i \le k$, have already been determined, then an upper bound for z can be obtained by relaxing the requirement $x_i = 0$ or 1

Scanned with CamScanner

```
Algorithm Bound (cp, cw, k)
{
b := cp; c := cw;
for i := k+1 to n do
  {
    c := c + w[i];
    if (c < m) then
      b := b + p[i];
    else
      return b + (1 - (c - m) / w[i]) * p[i];
  }
return b;
}
Algorithm Bknap(k, cp, cw)
{
// Generate left child
if (cw + w[k] <= m) then
{
y[k] := 1;
if (k < n) then Bknap (k+1, cp + p[k], cw + w[k]);
if ((cp + p[k] > fp) and (k = n)) then
  {
    fp := cp + p[k]; fw := cw + w[k];
    for j := 1 to k do
      x[j] := y[j];
  }
// Generate right child
if (Bound (cp, cw, k) >= fp) then
  { y[k] := 0;
    if (k < n) then Bknap (k+1, cp, cw);
    if ((cp > fp) and (k = n)) then
      {
```

fp := cp; fw := cw;
for j := 1 to k do
x[j] := y[j];
}

18/48

Scanned with CamScanner