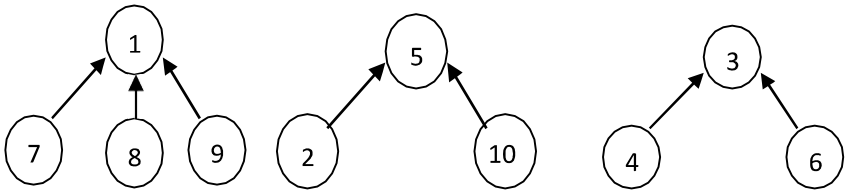
UNIT II:

Disjoint set operations, union and find algorithms, AND/OR graphs, Connected Components and Spanning trees, Bi-connected components

Backtracking-General method, applications- The 8-queen problem, sum of subsets problem, graph coloring, Hamiltonian cycles.

Sets and Disjoint Set Union:

Disjoint Set Union: Considering a set S={1,2,3…10} (when n=10), then elements can be partitioned into three disjoint sets s1={1,7,8,9},s2={2,5,10} and s3={3,4,6}. Possible tree representations are:



In this representation each set is represented as a tree. Nodes are linked from the child to parent rather than usual method of linking from parent to child.

The operations on these sets are:

1. Disjoint set union
2. Find(i)
3. Min Operation
4. Delete
5. Intersect

1. Disjoint Set union:

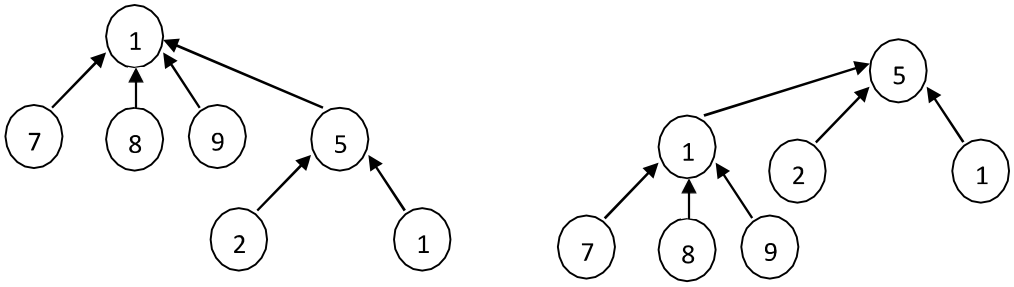
If S_i and S_j are two disjoint sets, then their union S_i U S_j = all the elements x such that x is in S_i or S_j. Thus S_1 U S_2 ={1,7,8,9,2,5,10}.

2. Find(i):

Given the element I, find the set containing i. Thus, 4 is in set S_3, 9 is in S_1.

UNION operation:

Union(i,j) requires two tree with roots i and j be joined. S_1 U S_2 is obtained by making any one of the sets as sub tree of other.



Simple Algorithm for Union:

```
Algorithm Union(i,j)
{
//replace the disjoint sets with roots i and j, I not equal to j by their
union Integer i,j;
P[j] :=i;
}
```

Example:

Implement following sequence of operations Union(1,3),Union(2,5),Union(1,2)

Solution:

Initially parent array contains zeros.

0	0	0	0	0	0
1	2	3	4	5	6

1. After performing union(1,3)operation Parent[3]:=1

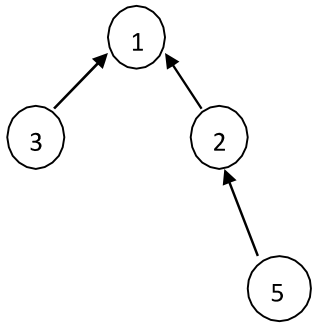
0	0	1	0	0	0
1	2	3	4	5	6

2. After performing union(2,5)operation Parent[5]:=2

0	0	1	0	2	0
1	2	3	4	5	6

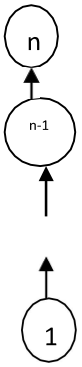
3. After performing union(1,2)operation Parent[2]:=1

0	1	1	0	2	0
1	2	3	4	5	6



Process the following sequence of union operations Union(1,2),Union(2,3) Union(n-1,n)

Degenerate Tree:



The time taken for n-1 unions is O(n).

Find(i) operation: determines the root of the tree containing

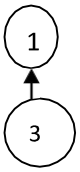
element i. Simple Algorithm for Find:

Algorithm Find(i)

```
{
    j:=i;
    while(p[j]>0) do
    j:=p[j]; return j;
}
```

Find Operation: Find(i) implies that it finds the root node of i^{th} node, in other words it returns the name of the set i.

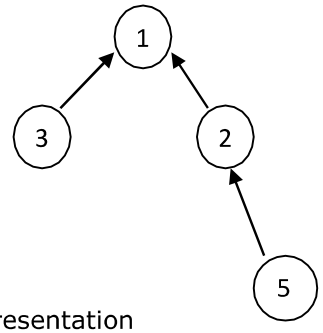
Example: Consider the Union(1,3)



Find(1)=0
Find(3)=1, since its parent is 1. (i.e, root is 1)

Example:

Considering



Array Representation

P[i]	0	1	1	2
i	1	2	3	5

Find(5)=1
Find(2)=1
Find(3)=1

The root node represents all the nodes in the tree. Time Complexity of 'n' find operations is $O(n^2)$.

To improve the performance of union and find algorithms by avoiding the creation of degenerate tree. To accomplish this, we use weighting rule for Union(i,j).

Weighting Rule for Union(i,j)

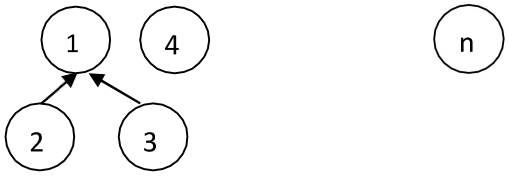
```
1   Algorithm WeightedUnion(i, j)
2   // Union sets with roots i and j, i ≠ j, using the
3   // weighting rule. p[i] = -count[i] and p[j] = -count[j].
4   {
5       temp := p[i] + p[j];
6       if (p[i] > p[j]) then
7       { // i has fewer nodes.
8           p[i] := j; p[j] := temp;
9       }
10      else
11      { // j has fewer or equal nodes.
12          p[j] := i; p[i] := temp;
13      }
14  }
```

Union algorithm with weighting rule

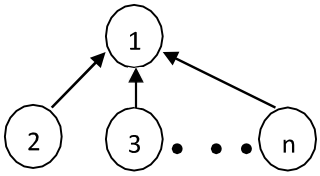
Tree obtained with
weighted Initially

Union(1,2)

Union(1,3)



Union(1,n)



Collapsing Rule for Find(i)

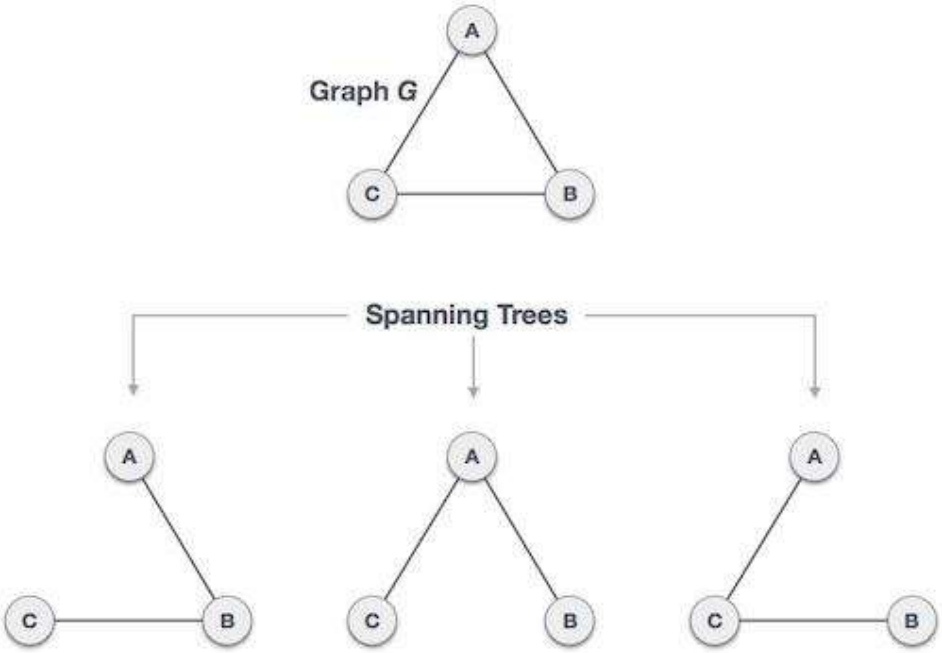
```
1   Algorithm CollapsingFind(i)
2   // Find the root of the tree containing element i. Use the
3   // collapsing rule to collapse all nodes from i to the root.
4   {
5       r := i;
6       while (p[r] > 0) do r := p[r]; // Find the root.
7       while (i ≠ r) do  // Collapse nodes from i to root r.
8       {
9           s := p[i]; p[i] := r; i := s;
10      }
11      return r;
12  }
```

Find algorithm with collapsing rule

Spanning Trees:

A spanning tree is a subset of Graph G, which has all the vertices covered with minimum possible number of edges. Hence, a spanning tree does not have cycles and it cannot be disconnected..

By this definition, we can draw a conclusion that every connected and undirected Graph G has at least one spanning tree. A disconnected graph does not have any spanning tree, as it cannot be spanned to all its vertices.



We found three spanning trees off one complete graph. A complete undirected graph can have maximum $\mathbf{n^{n-2}}$ number of spanning trees, where **n** is the number of nodes. In the above addressed example, **n is 3,** hence $\mathbf{3^{3-2} = 3}$spanning trees are possible.

General Properties of Spanning Tree

We now understand that one graph can have more than one spanning tree. Following are a few properties of the spanning tree connected to graph G −

- A connected graph G can have more than one spanning tree.
- All possible spanning trees of graph G, have the same number of edges and vertices.
- The spanning tree does not have any cycle (loops).
- Removing one edge from the spanning tree will make the graph disconnected, i.e. the spanning tree is **minimally connected**.
- Adding one edge to the spanning tree will create a circuit or loop, i.e. the spanning tree is **maximally acyclic**.

Mathematical Properties of Spanning Tree

- Spanning tree has **n-1** edges, where **n** is the number of nodes (vertices).
- From a complete graph, by removing maximum **e - n + 1** edges, we can construct a spanning tree.
- A complete graph can have maximum $\mathbf{n^{n-2}}$ number of spanning trees.

Thus, we can conclude that spanning trees are a subset of connected Graph G and disconnected graphs do not have spanning tree.

Application of Spanning Tree

Spanning tree is basically used to find a minimum path to connect all nodes in a graph. Common application of spanning trees are −

- Civil Network Planning
- Computer Network Routing Protocol
- Cluster Analysis

AND/OR GRAPH:

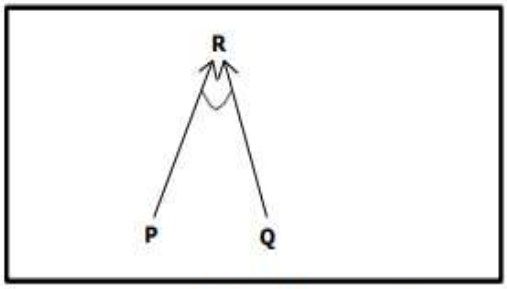
And/or graph is a specialization of hypergraph which connects nodes by sets of arcs rather than by a single arcs. A hypergraph is defined as follows:

A hypergraph consists of: N, a set of nodes,

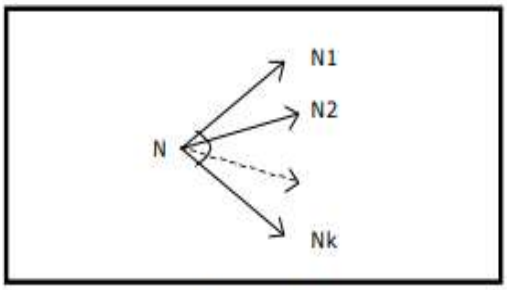
H, a set of hyperarcs defined by ordered pairs, in which the first implement of the pair is a node of N and the second implement is the subset of N.

An ordinary graph is a special case of hypergraph in which all the sets of decendent nodes have a cardinality of 1.

Hyperarcs also known as K-connectors, where K is the cardinality of the set of decendent nodes. If K = 1, the descendent may be thought of as an OR nodes. If K > 1, the elements of the set of decendents may be thought of as AND nodes. In this case the connector is drawn with individual edges from the parent node to each of the decendent nodes; these individual edges are then joined with a curved link. And/or graph for the expression P and Q -> R is follows:



Expression for P and Q -> R



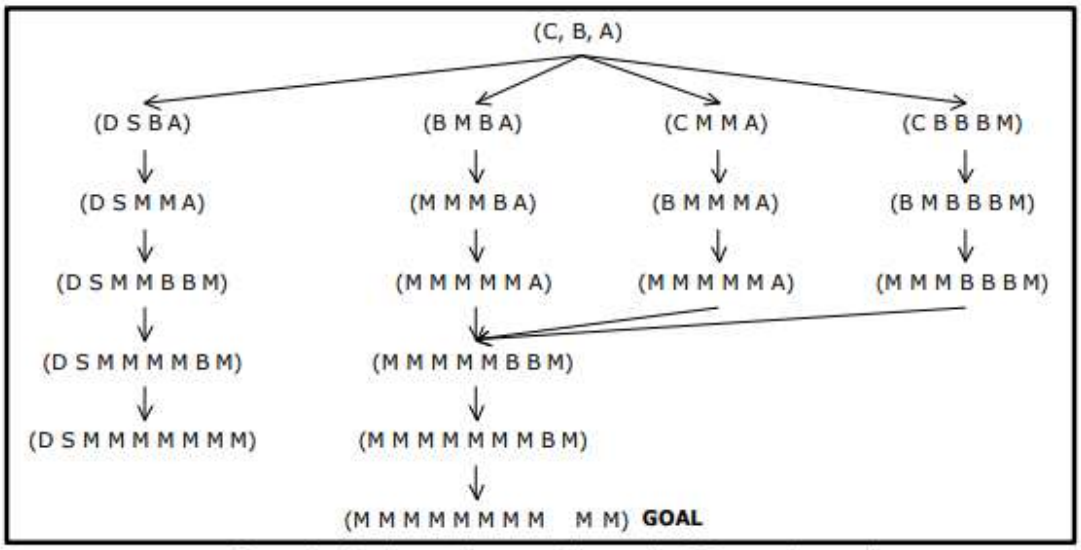
A K-Connector

The K-connector is represented as a fan of arrows with a single tie is shown above. The and/or graphs consists of nodes labelled by global databases. Nodes labelled by compound databases have sets of successor nodes. These successor nodes are called AND nodes, in order to process the compound database to termination, all the compound databases must be processed to termination. For example consider, consider a boy who collects stamps (M). He has for the purpose of exchange a winning conker (C), a bat (B) and a small toy animal (A). In his class there are friends who are also keen collectors of different items and will make the following exchanges.

1. 1 winning conker (C) for a comic (D) and a bag of sweets (S).
2. 1 winning conker (C) for a bat (B) and a stamp (M).
3. 1 bat (B) for two stamps (M, M).
4. 1 small toy animal (A) for two bats (B, B) and a stamp (M).

The problem is how to carry out the exchanges so that all his exchangable items are converted into stamps (M). This task can be expressed more briefly as:

1. Initial state = (C, B, A)
2. Transformation rules:
   a. If C then (D, S)
   b. If C then (B, M)
   c. If B then (M, M)
   d. If A then (B, B, M)
3. The goal state is to left with only stamps (M, ... , M)



Expansion for the exchange problem using OR connectors only

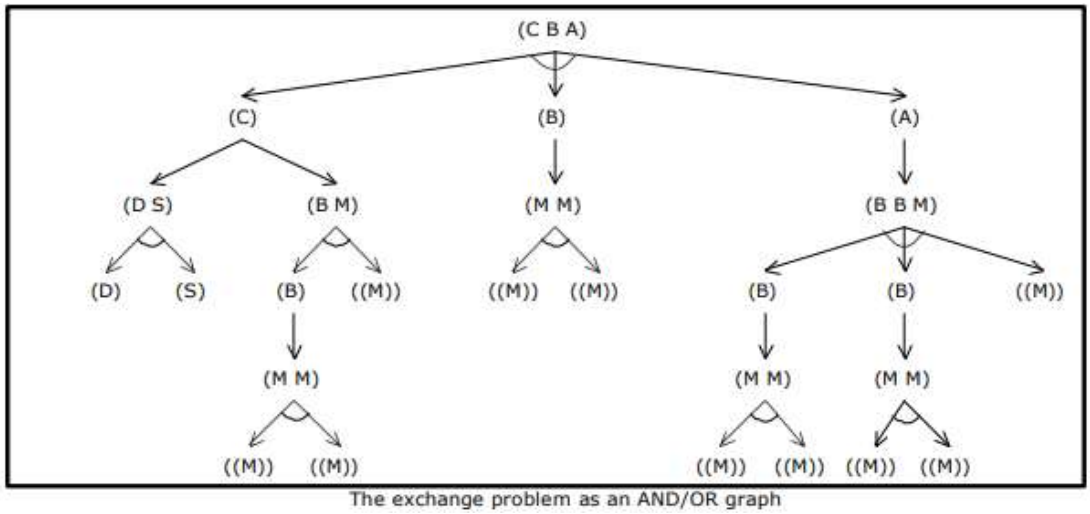
The figure shows that, a lot of extra work is done by redoing many of the transformations. This repetition can be avoided by decomposing the problem into subproblems. There are two major ways to order the components:

1. The components can either be arranged in some fixed order at the time they are generated (or).
2. They can be dynamically reordered during processing.

The more flexible system is to reorder dynamically as the processing unfolds. It can be represented by and/or graph. The solution to the exchange problem will be:

Swap conker for a bat and a stamp, then exchange this bat for two stamps. Swap his own bat for two more stamps, and finally swap the small toy animal for two bats and a stamp. The two bats can be exchanged for two stamps.

The previous exchange problem, when implemented as an and/or graph looks as follows:

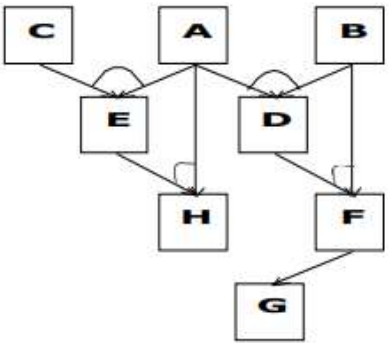


The exchange problem as an AND/OR graph

Example 1:

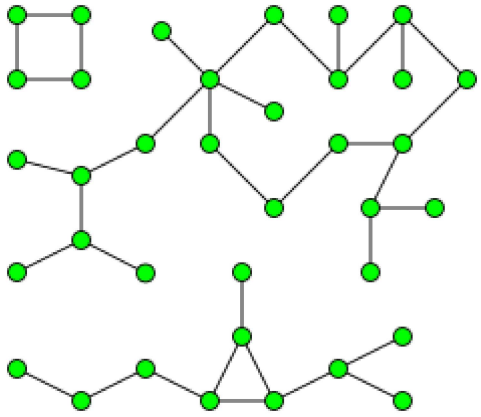
Draw an AND/OR graph for the following prepositions:

1. A
2. B
3. C
4. A ^ B -> D
5. A ^ C -> E
6. B ^ D -> F
7. F -> G
8. A ^ E -> H



Connected components

In graph theory, a connected component (or just component) of an undirected graph is a subgraph in which any two vertices are connected to each other by paths, and which is connected to no additional vertices in the super graph. For example, the graph shown in the illustration has three connected components. A vertex with no incident edges is itself a connected component. A graph that is itself connected has exactly one connected component, consisting of the whole graph.



A graph with three connected components.

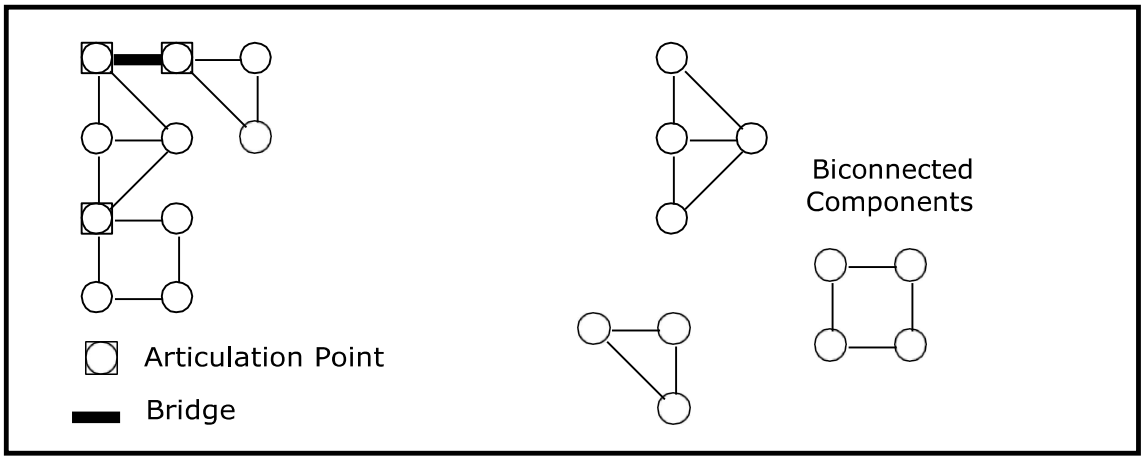
Biconnected Components:

Let G = (V, E) be a connected undirected graph. Consider the following definitions:

Articulation Point (or Cut Vertex): An articulation point in a connected graph is a vertex (together with the removal of any incident edges) that, if deleted, would break the graph into two or more pieces..

Bridge: Is an edge whose removal results in a disconnected graph.

Biconnected: A graph is biconnected if it contains no articulation points. In a biconnected graph, two distinct paths connect each pair of vertices. A graph that is not biconnected divides into biconnected components. This is illustrated in the following figure:



Articulation Points and Bridges

Biconnected graphs and articulation points are of great interest in the design of network algorithms,

because these are the "critical" points, whose failure will result in the network becoming disconnected.

Let us consider the typical case of vertex v, where v is not a leaf and v is not the root. Let w_1, w_2, ... w_k be the children of v. For each child there is a subtree of the DFS tree rooted at this child. If for some child, there is no back edge going to a proper ancestor of v, then if we remove v, this subtree becomes disconnected from the rest of the graph, and hence v is an articulation point.

L (u) = min {DFN (u), min {L (w) | w is a child of u}, min {DFN (w) | (u, w) is a back edge}}.

L (u) is the lowest depth first number that can be reached from 'u' using a path of descendents followed by at most one back edge. It follows that, If 'u' is not the root then 'u' is an articulation point iff 'u' has a child 'w' such that:

L (w) ≥ DFN (u)

Algorithm for finding the Articulation points:

Pseudocode to compute DFN and L.

Algorithm Art (u, v)

```
// u is a start vertex for depth first search. V is its parent if any in the depth first
// spanning tree. It is assumed that the global array dfn is initialized to zero and
that // the global variable num is initialized to 1. n is the number of vertices in G.
{
        dfn [u] := num; L [u] := num; num
        := num + 1; for each vertex w
        adjacent from u do
        {
                if (dfn [w] = 0) then
                {
```

```
                    Art  (w,  u);                    // w
                    is unvisited. L [u] := min (L [u], L
                    [w]);
                }
        else if (w ≠ v) then L [u] := min (L [u], dfn [w]);
        }
}
```

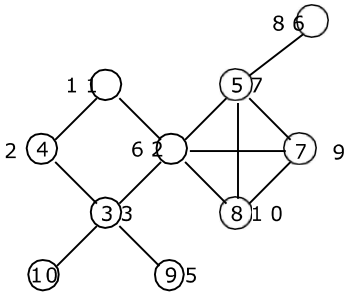
Algorithm for finding the Biconnected Components:

Algorithm BiComp (u, v)

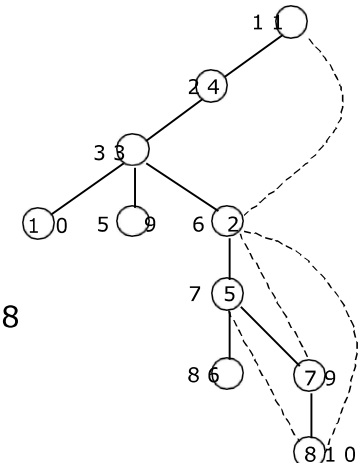
```
// u is a start vertex for depth first search. V is its parent if any in the depth first
// spanning tree. It is assumed that the global array dfn is initially zero and that the
// global variable num is initialized to 1. n is the number of vertices in G.
{
        dfn [u] := num; L [u] := num; num
        := num + 1; for each vertex w
        adjacent from u do
        {
                if ((v ≠ w) and (dfn [w] ≤ dfn
                        [u])) then add (u, w)
                        to the top of a stack
                        s;
                if (dfn [w] = 0) then
                {
                        if (L [w] ≥ dfn [u]) then
                        {
                                write ("New
                                bicomponent");
                                repeat
                                {
                                        Delete an edge from the
                                        top of stack s; Let this
                                        edge be (x, y);
                                  Write (x, y);
                                } until (((x, y) = (u, w)) or ((x, y) = (w, u)));
                        }
                        BiComp  (w,  u);     // w is unvisited. L [u] := min (L [u], L [w]);
                }
                else if (w ≠ v) then L [u] : = min (L [u], dfn [w]);
        }
}
```

Example:

For the following graph identify the articulation points and Biconnected components:



4



4 Graph Dept h First Spanning Tree

To identify the articulation points, we use:

L (u) = min {DFN (u), min {L (w) | w is a child of u}, min {DFN (w) | w is a vertex to which there is back edge from u}}

L (1) = min {DFN (1), min {L (4)}} = min {1, L (4)} = min {1, 1} = 1

L (4) = min {DFN (4), min {L (3)}} = min {2, L (3)} = min {2, 1} = 1

L (3) = min {DFN (3), min {L (10), L (9), L (2)}} =
= min {3, min {L (10), L (9), L (2)}} = min {3, min {4, 5, 1}} = 1

L (10) = min {DFN (10)} = 4

L (9) = min {DFN (9)} = 5

L (2) = min {DFN (2), min {L (5)}, min {DFN (1)}}
= min {6, min {L (5)}, 1} = min {6, 6, 1} = 1

L (5) = min {DFN (5), min {L (6), L (7)}} = min {7, 8, 6} = 6

L (6) = min {DFN (6)} = 8
L (7) = min {DFN (7), min {L (8}, min {DFN (2)}}
= min {9, L (8) , 6} = min {9, 6, 6} = 6

L (8) = min {DFN (8), min {DFN (5), DFN (2)}}
= min {10, min (7, 6)} = min {10, 6} = 6

Therefore, L (1: 10) = (1, 1, 1, 1, 6, 8, 6, 6, 5, 4)

Finding the Articulation Points:

Vertex 1: Vertex 1 is not an articulation point. It is a root node. Root is an articulation point if it has two or more child nodes.

Vertex 2: is an articulation point as child 5 has L (5) = 6 and DFN (2) = 6, So, the condition L (5) = DFN (2) is true.

Vertex 3: is an articulation point as child 10 has L (10) = 4 and DFN (3) = 3, So, the condition L (10) > DFN (3) is true.

Vertex 4: is not an articulation point as child 3 has L (3) = 1 and DFN (4) = 2, So, the condition L (3) $\geq$ DFN (4) is false.

Vertex 5: is an articulation point as child 6 has L (6) = 8, and DFN (5) = 7, So, the condition L (6) > DFN (5) is true.

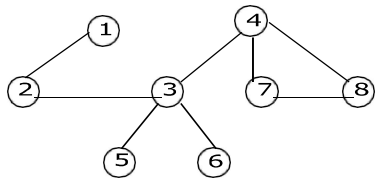
Vertex 7: is not an articulation point as child 8 has L (8) = 6, and DFN (7) = 9, So, the condition L (8) $\geq$ DFN (7) is false.

Vertex 6, Vertex 8, Vertex 9 and Vertex 10 are leaf nodes.

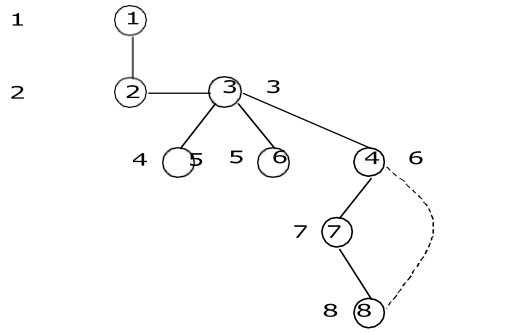
Therefore, the articulation points are {2, 3, 5}.

Example:

For the following graph identify the articulation points and Biconnected components:



Graph



DFS spanningTree

L (u) = min {DFN (u), min {L (w) | w is a child of u}, min {DFN (w) | w is a vertex to which there is back edge from u}}

L (1) = min {DFN (1), min {L (2)}} = min {1, L (2)} = min {1, 2} = 1

L (2) = min {DFN (2), min {L (3)}} = min {2, L (3)} = min {2, 3} = 2

L (3) = min {DFN (3), min {L (4), L (5), L (6)}} = min {3, min {6, 4, 5}} = 3

L (4) = min {DFN (4), min {L (7)} = min {6, L (7)} = min {6, 6} = 6

L (5) = min {DFN (5)} = 4

L (6) = min {DFN (6)} = 5

L (7) = min {DFN (7), min {L (8)}} = min {7, 6} = 6

L (8) = min {DFN (8), min {DFN (4)}} = min {8, 6} = 6

Therefore, L (1: 8) = {1, 2, 3, 6, 4, 5, 6, 6}

Finding the Articulation Points:

Check for the condition if L (w) $\geq$ DFN (u) is true, where w is any

child ofu. Vertex 1: Vertex 1 is not an articulation point.
It is a root node. Root is an articulation point if it has two or more child nodes.

Vertex 2: is an articulation point as L (3) = 3 and DFN (2) = 2.
So, the condition is true

Vertex 3: is an articulation Point as:
I. L (5) = 4 and DFN (3) = 3

II. L (6) = 5 and DFN (3) = 3 and
III. L (4) = 6 and

DFN (3) = 3 So, the

condition true in above

cases

Vertex 4: is an articulation point as L (7) = 6 and DFN (4) = 6.
So, the condition is true

Vertex 7: is not an articulation point as L (8) = 6 and DFN (7) = 7.
So, the condition is False

Vertex 5, Vertex 6 and Vertex 8 are leaf

nodes. Therefore, the articulation points

are {2, 3, 4}.