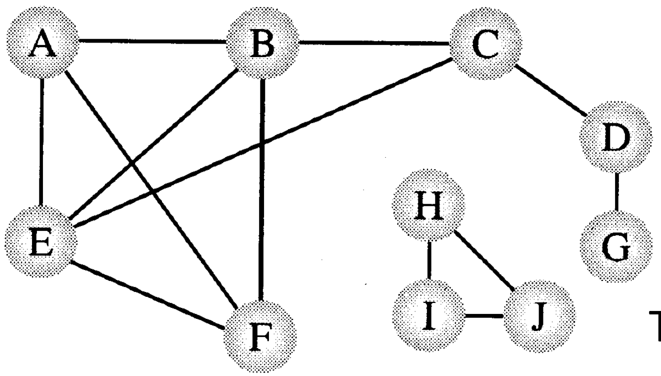
CSE 2202
Design and Analysis of Algorithms – I
Lecture 3:
Biconnected Components, Articulation Points and Bridge

Connectivity/Biconnectivity for Undirected Graph

- A node and all the nodes reachable from it compose a **connected component**.
  - A graph is called **connected** if it has only one connected component.

Since the function **visit**() of DFS visits every node that is reachable and has not already been visited, the DFS can easily be modified to print out the connected components of a graph.



Two connected components

Connectivity/Biconnectivity for Undirected Graph

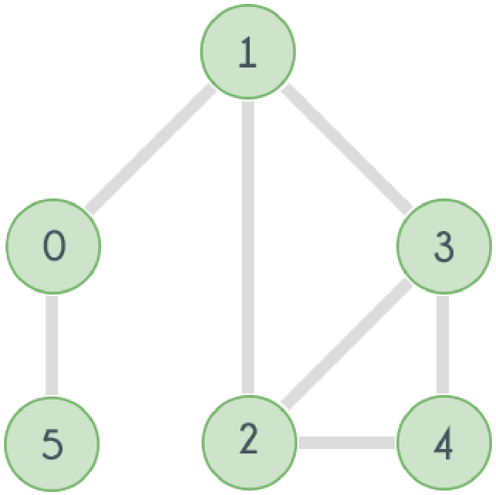


Fig. 1

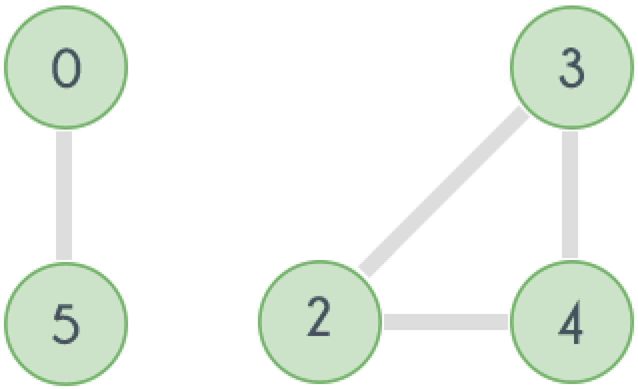


Fig. 2

Connectivity/Biconnectivity

- In actual uses of graphs, such as networks, we need to establish not only that every node is connected to every other node, but also there are **at least two independent paths between any two nodes**.

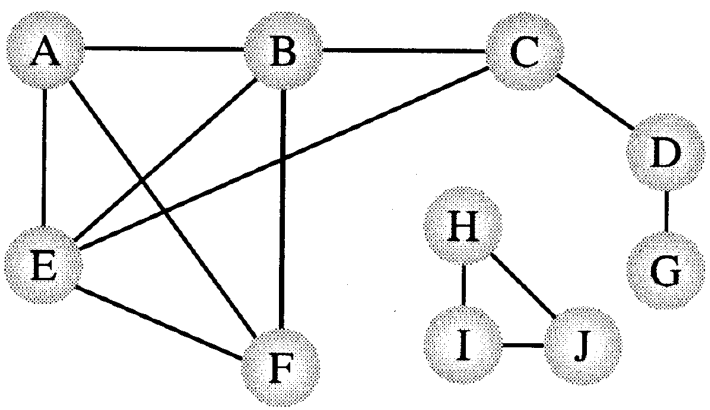
- A maximum set of nodes for which there are two different paths is called **biconnected components.**

Connectivity/Biconnectivity

A graph is deemed biconnected if it meets the following criteria:

It exhibits connectivity, which means there is a simple path that allows for travel from any vertex to any other vertex within the graph.

The graph maintains its connectivity even when any single vertex is removed.



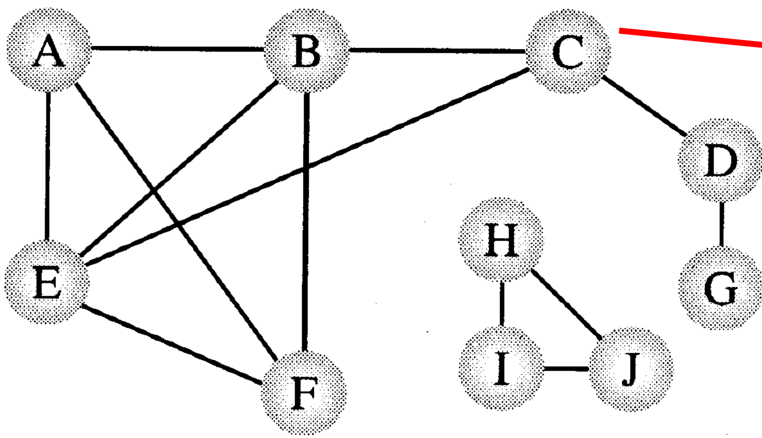
{H,I,J} and {A,B,C,E,F} are biconnected.

Connectivity/Biconnectivity

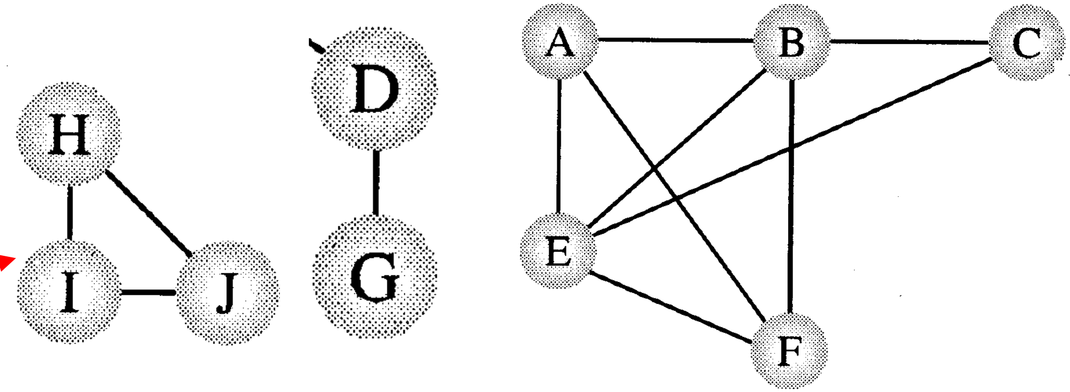
- Another way to define this concept is that there are **no single points of failure, that is -**
  - no nodes that when deleted along with any adjoining arcs, would split the graph into two or more separate connected components.
  - Such a node is called an **articulation point**.

Connectivity/Biconnectivity

- If a graph contains no articulation points, then it is biconnected.
  - If a graph does contain articulation points, then it is useful to split the graph into the pieces where each piece is a maximal biconnected subgraph called a **biconnected component**.



Articulation points



Three biconnected components

Connectivity/Biconnectivity

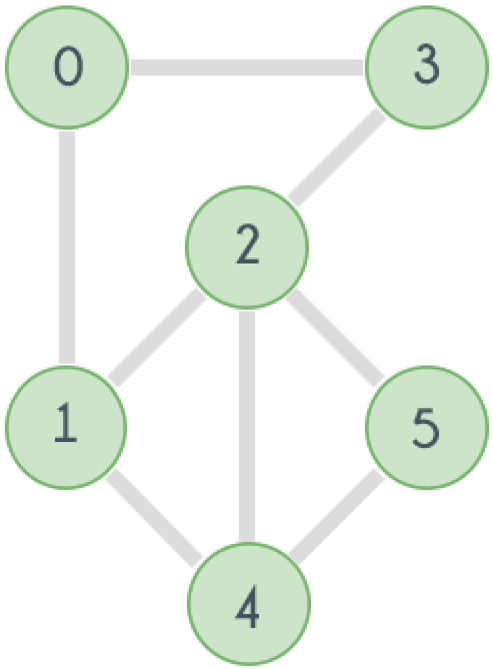


Fig. 1

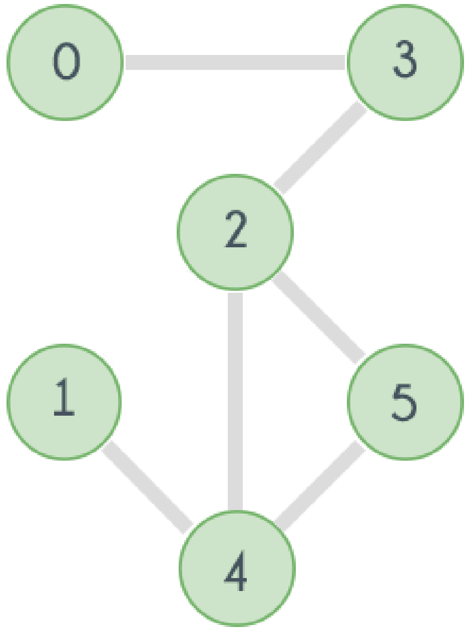


Fig. 2

How to Find Bridges

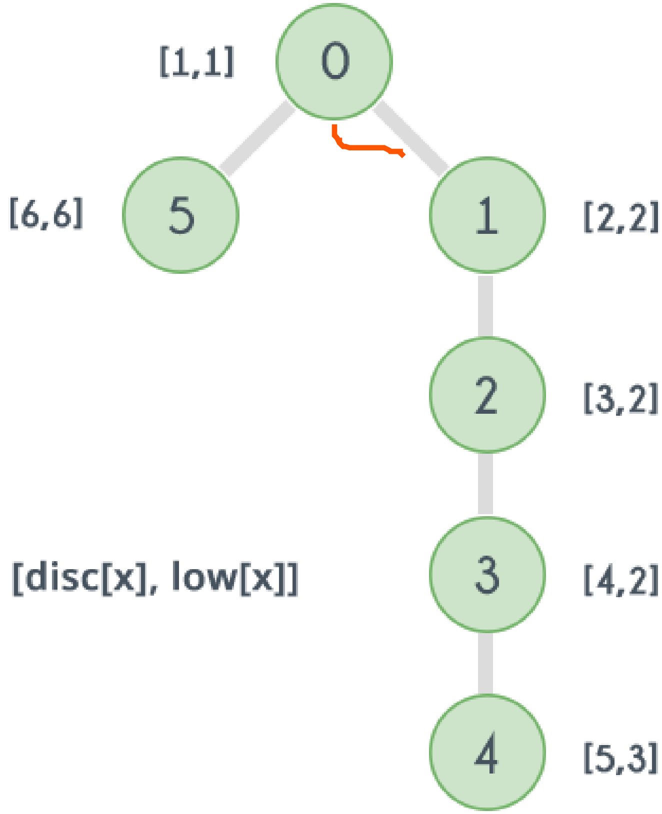


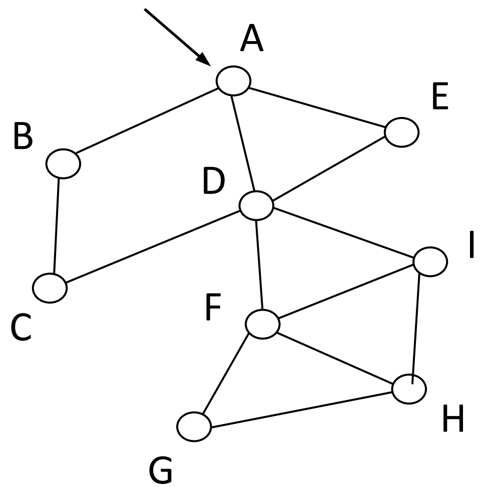
Fig. 4

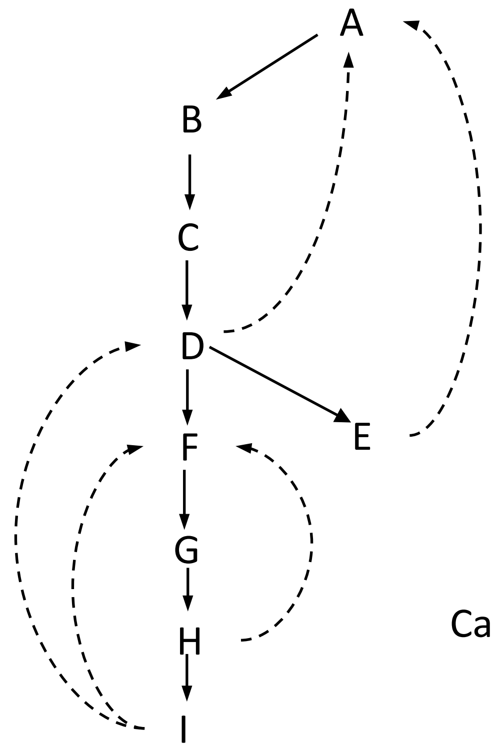
Finding Articulation Points

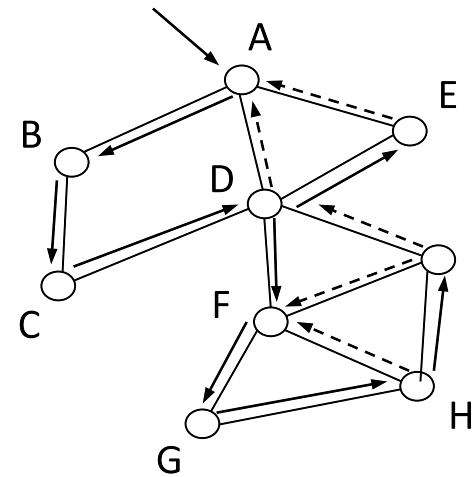
- Problem:
  - Given any graph $G = (V, E)$, find all the articulation points.
  - Possible strategy:
    - For all vertices v in V:
      - Remove v and its incident edges
      - Test connectivity using a DFS.
    - Execution time: $\Theta(n(n+m))$.
    - Can we do better?

Finding Articulation Points

- A DFS tree can be used to discover articulation points in $\Theta(n + m)$ time.

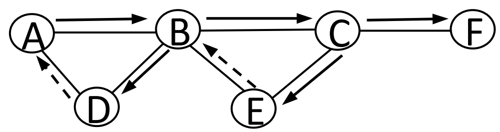




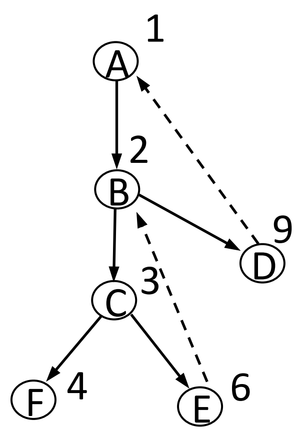


Can you characterize *D* ?

Depth First Search number

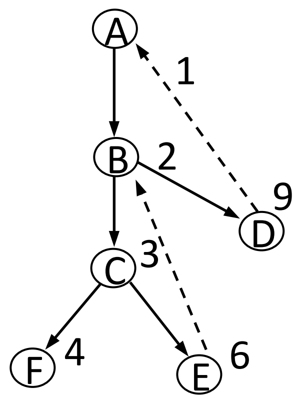


$G = (V, E)$



A	B	C	D	E	F
1	2	3	9	6	4

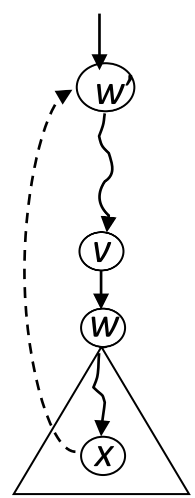
Any relation between Discovery time and articulation point ?



Assume that $(a,b) \Leftrightarrow a \rightarrow b$

Tree edge : (a,b) $a < b$

Back edge : (a,b) $a > b$



If there is a back edge from x to a proper ancestor of v, then v is reachable from x.

Finding Articulation Points

- A DFS tree can be used to discover articulation points in $\boxed{\Theta(n + m)}$ time.
  - We start with a program that computes a DFS tree labeling the vertices with their discovery times.
  - We also compute a function called low(v) that can be used to characterize each vertex as an articulation or non-articulation point.
  - The root of the DFS tree will be treated as a special case:
    - The root has a d[] value of 1.

Definition of low(v)

- Definition. The value of *low*(v) is the discovery time of the vertex closest to the root and reachable from v by following zero or more tree edges downward, and then at most one back edge.
- We can efficiently compute *low* by performing a postorder traversal of the depth-first spanning tree.

low[v] = min{

$d[v]$,

lowest d[w] among all back edges (v, w)

lowest low[w] among all tree edges (v, w)

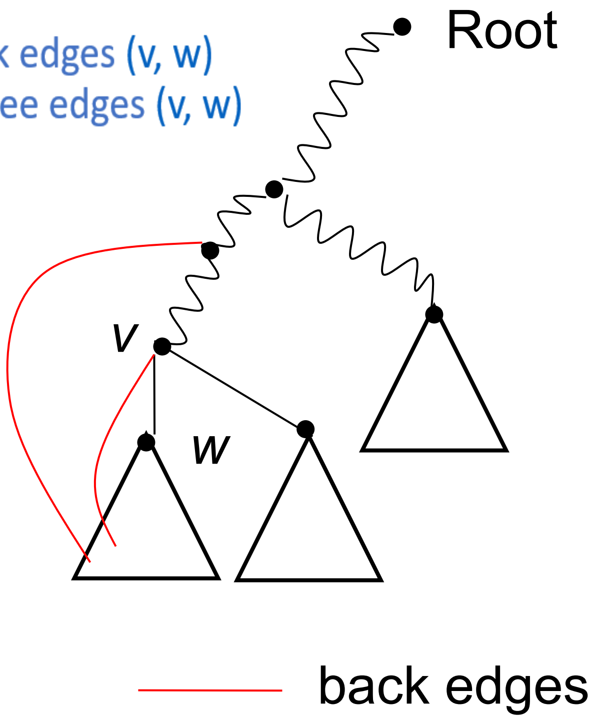
}

- low(v) < $d[v]$ indicates if there is another way to reach v which is not via its parent

Low(v)

- Observe that if there is a back edge from somewhere below v to above v in the tree, then low(v) < $d[v]$
- Otherwise low(v) = $d[v]$

low[v] = min{
  d[*v*],
  lowest d[w] among all back edges (v, w)
  lowest low[w] among all tree edges (v, w)
}



Low(v)

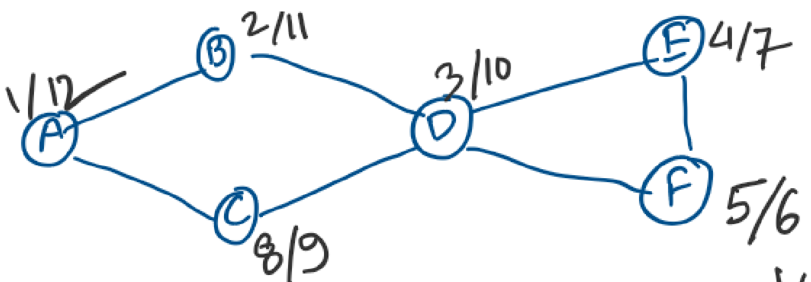
$low[v]$ = min{

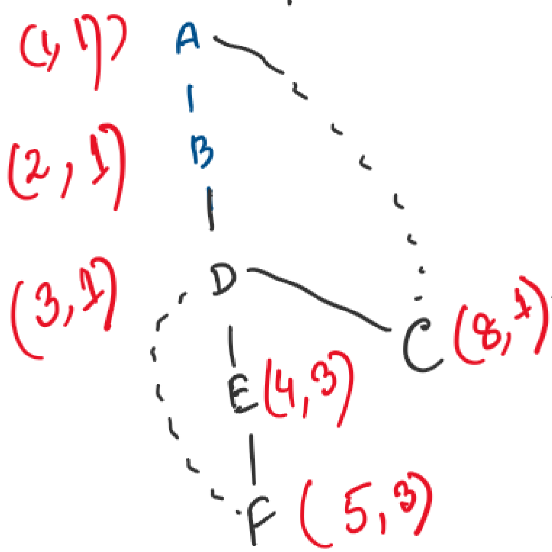
$d[v]$,

lowest $d[w]$ among all back edges (v, w)

lowest $low[w]$ among all tree edges (v, w)

}





Vertex	d( )	low( )
A	1	1
B	2	1
C	8	1
D	3	1
E	4	3
F	5	3

Finding Articulation Points

- Let v be a non-root vertex of the DFS tree T.
- Then v is an articulation point of G if and only if there is a child w of v with $low(w) >= d[v]$.

Articulation Points: Pseudocode

```
Data: color[V], time, prev[V],d[V], f[V], low[V]
```

```
DFS(G) // where prog starts
{
   for each vertex u ∈ V
   {
      color[u] = WHITE;
       prev[u]=NIL;
       low[u]=inf;
       f[u]=inf; d[u]=inf;
   }
   time = 0;
   for each vertex u ∈ V
     if (color[u] == WHITE)
         DFS_Visit(u);
}
```

Articulation Points: Pseudocode

```
DFS_Visit(v)
{ color[v]=GREY;time=time+1;d[v] = time;
  low[v]= d[v];
  for each w ∈ Adj[v]{
    if(color[w] == WHITE){
        prev[w]=u;
       DFS_Visit(w);
       if low[w] >= d[v]
            record that vertex v is an articulation
       if (low[w] < low[v]) low[v] := low[w];
    }
    else if w is not the parent of v then
         //--- (v,w) is a BACK edge
  `        if (d[w] < low[v]) low[v] := d[w];
  }
  color[v] = BLACK;  time = time+1;   f[v] = time;
}
```

Special Case

- When "v" is a root of the DFS tree, you have to check it manually.

Finding Articulation Points

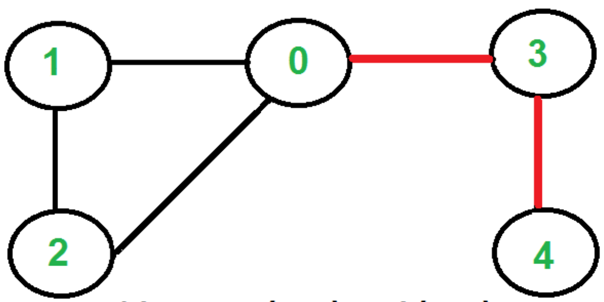
- The root of the DFS tree is an articulation point if and only if it has two or more children.
  - Suppose the root has two or more children.
    - Recall that back edges never link vertices between two different subtrees.
    - So, the subtrees are only linked through the root vertex and its removal will cause two or more connected components (i.e. the root is an articulation point).
  - Suppose the root is an articulation point.
    - This means that its removal would produce two or more connected components each previously connected to this root vertex.
    - So, the root has two or more children.

Source

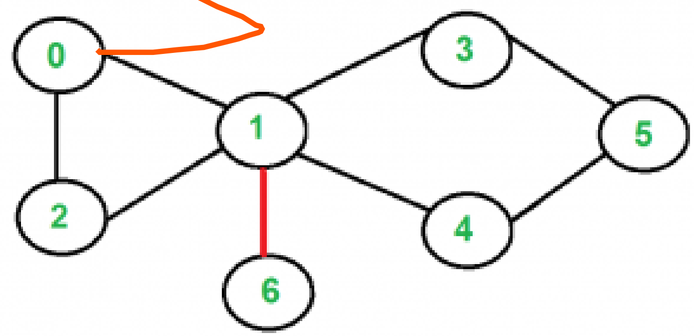
- Mark Allen Weiss – Data Structure and Algorithm Analysis in C
  - Articulation Point
- Exercise:
  - CLRS – Exercise 22-2

Bridge

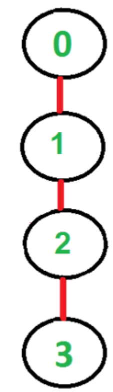
- An edge in an undirected connected graph is a bridge iff removing it disconnects the graph.
- For a disconnected undirected graph, definition is similar, a bridge is an edge removing which increases number of disconnected components.



Bridges are (0, 3) and (3, 4)

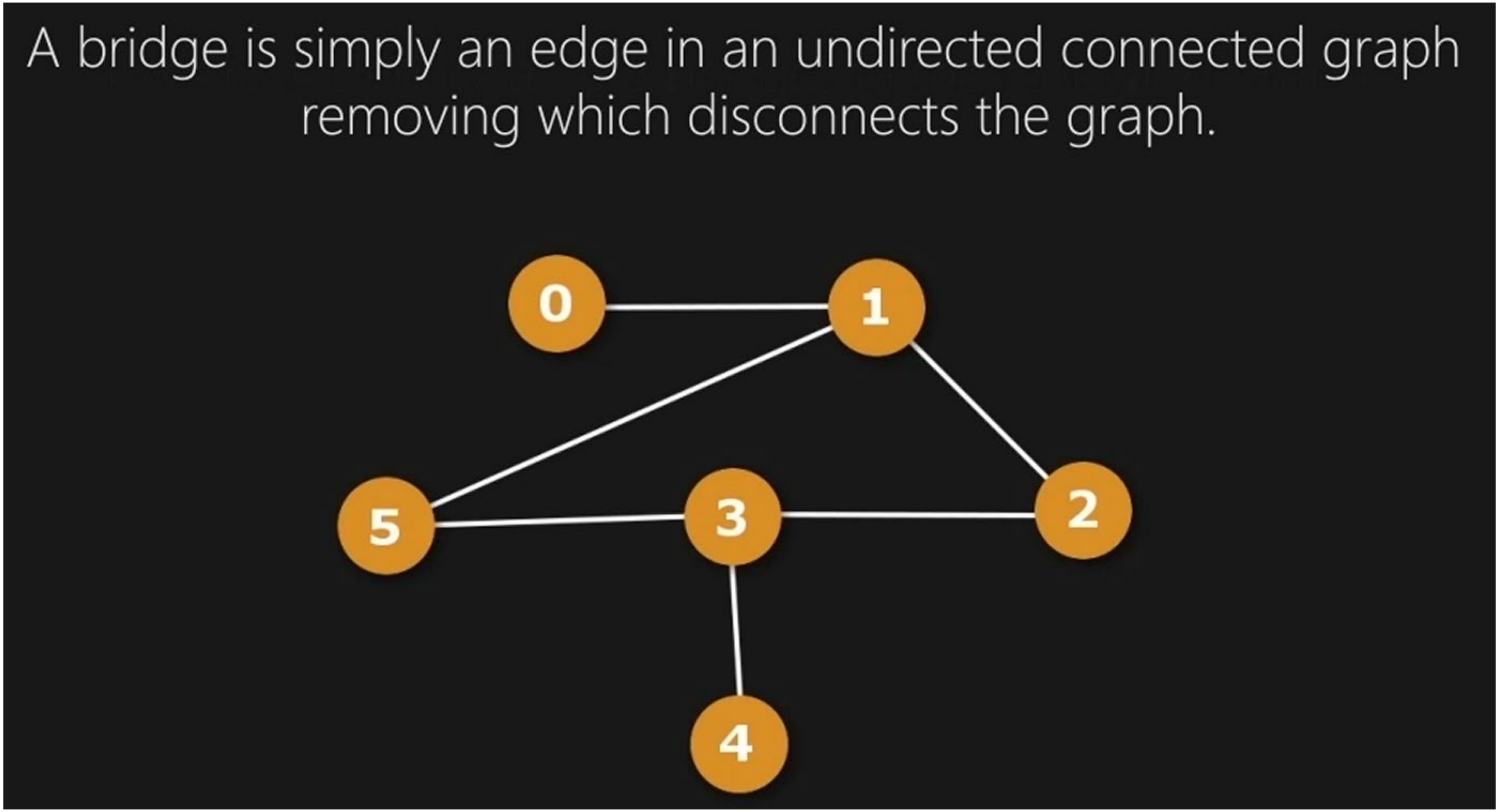


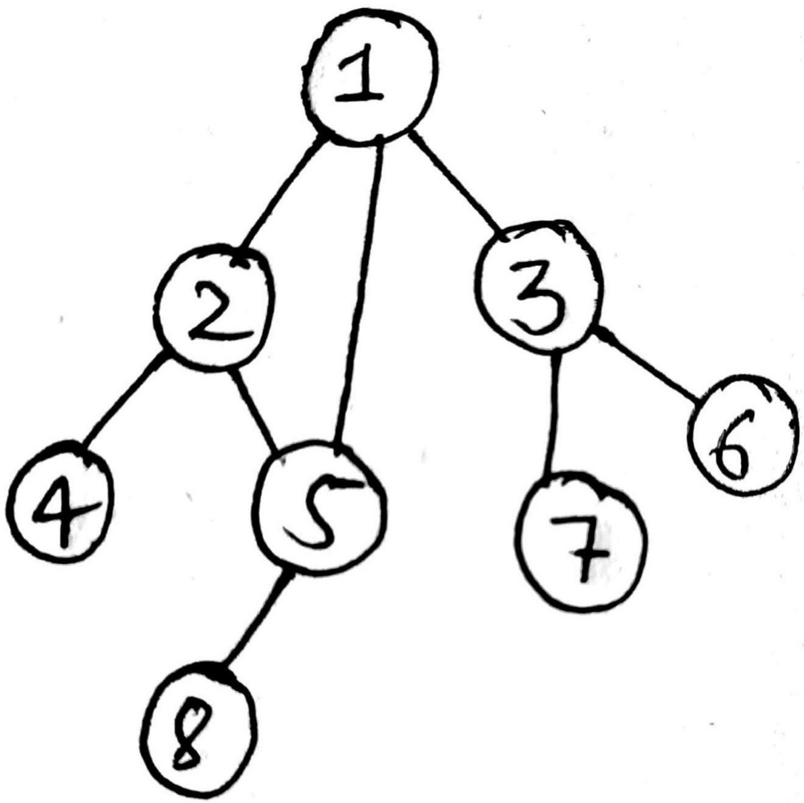
Bridge is (1, 6)

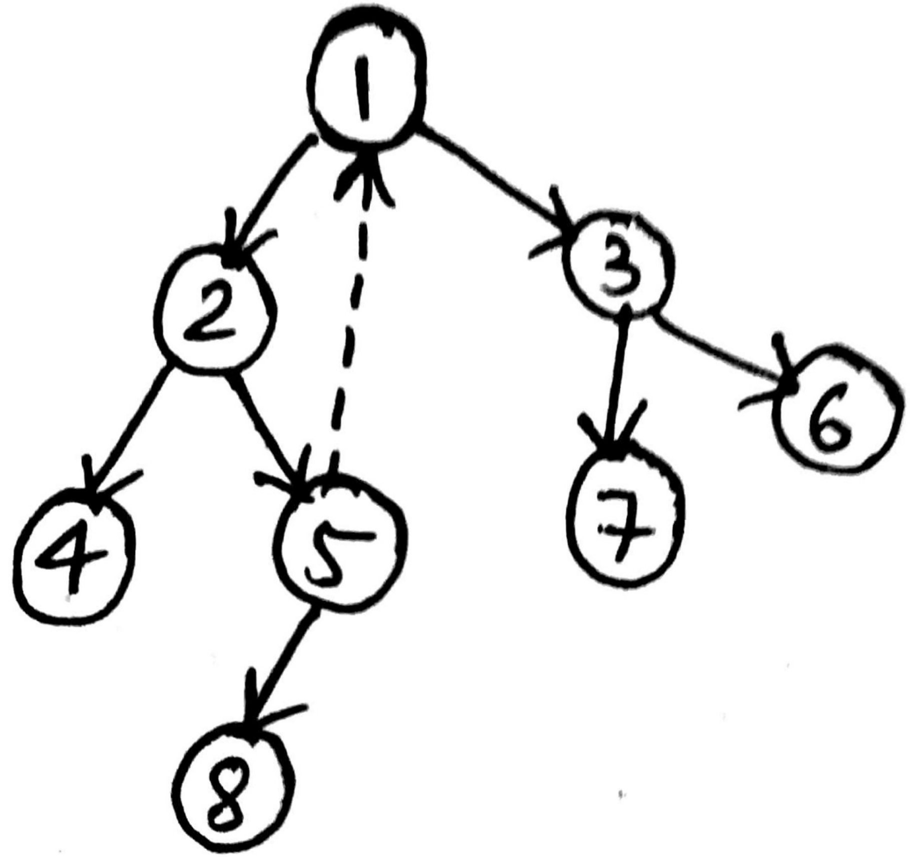


Bridges are (0,1), (1,2) and (2,3)

How to Find Bridges




1
2
3
4
5
7
6
8


1
2
3
4
5
7
6
8

How to Find Bridges

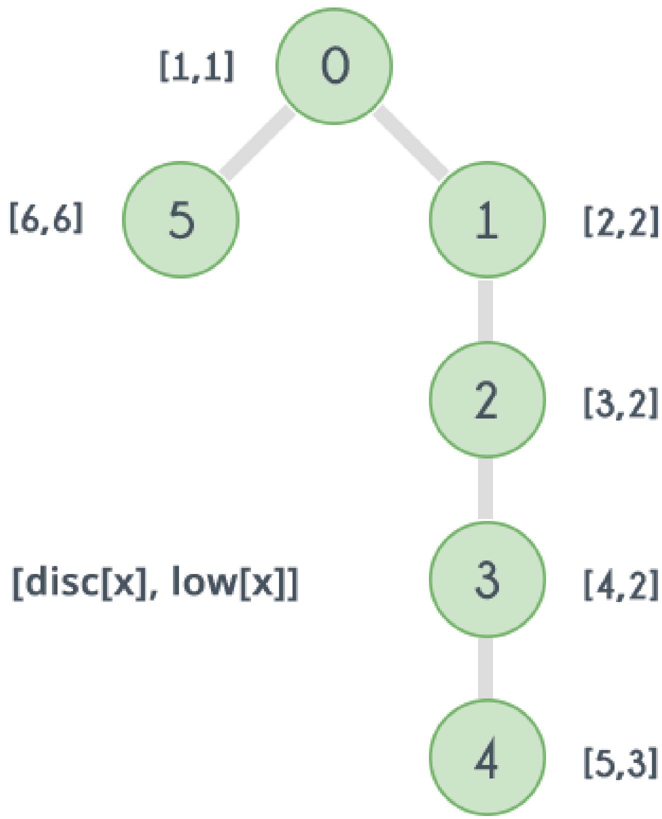


Fig. 4

Bridge: Pseudocode

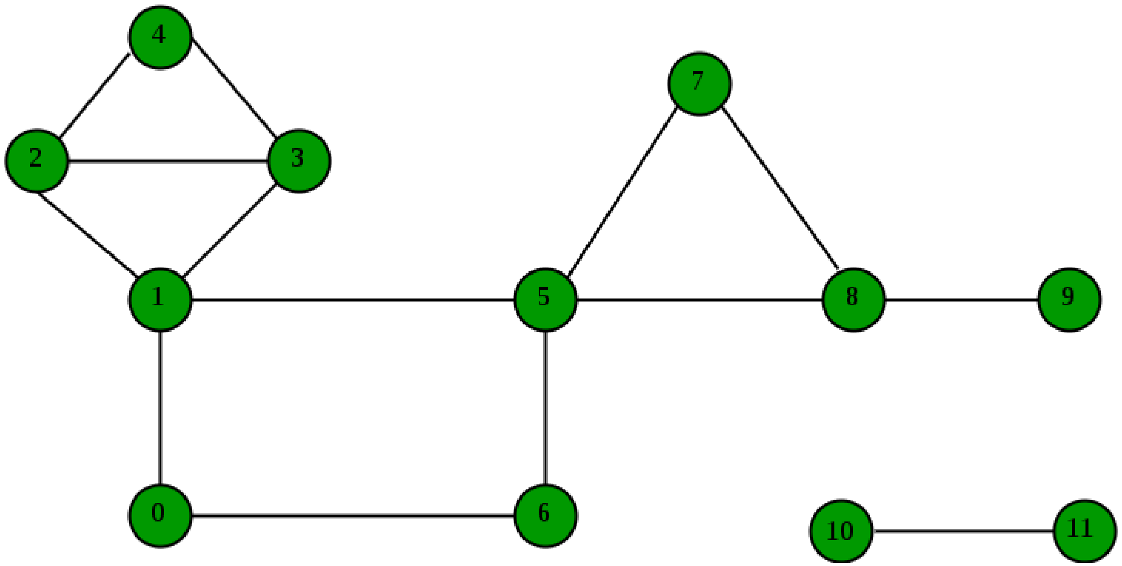
```
DFS_Visit(v)
{ color[v]=GREY; time=time+1; d[v] = time;
  low[v]= d[v];
  for each w ∈ Adj[v]{
    if(color[w] == WHITE){
          prev[w]=u;
        DFS_Visit(w);
        if low[w] > d[v]
               record that vertex (v, w) is a bridge
               (WHY??)
        if (low[w] < low[v]) low[v] := low[w];
    }
    else if w is not the parent of v then
         //--- (v,w) is a BACK edge
  `        if (d[w] < low[v]) low[v] := d[w];
  }
  color[v] = BLACK;  time = time+1;   f[v] = time;
}
```

Biconnected Components (BCC)

- **Biconnected Graph**: An undirected graph is called Biconnected if there are two vertex-disjoint paths between any two vertices. In a Biconnected Graph, there is a simple cycle through any two vertices.
- **Biconnected Componets:** A biconnected component is a maximal biconnected subgraph

- In graph, following are the bcc(s):

- 4–2 3–4 3–1 2–3 1–2
- 8–9
- 8–5 7–8 5–7
- 6–0 5–6 1–5 0–1
- 10–11



Biconnected Components (BCC)