CSCI203

Algorithms and Data Structures

Graphs – Articulation Points

Lecturer: Dr. Xueqiao Liu
Room 3.117
Email: xueqiao@uow.edu.au

Recall - $DFS\_ALL(G)$

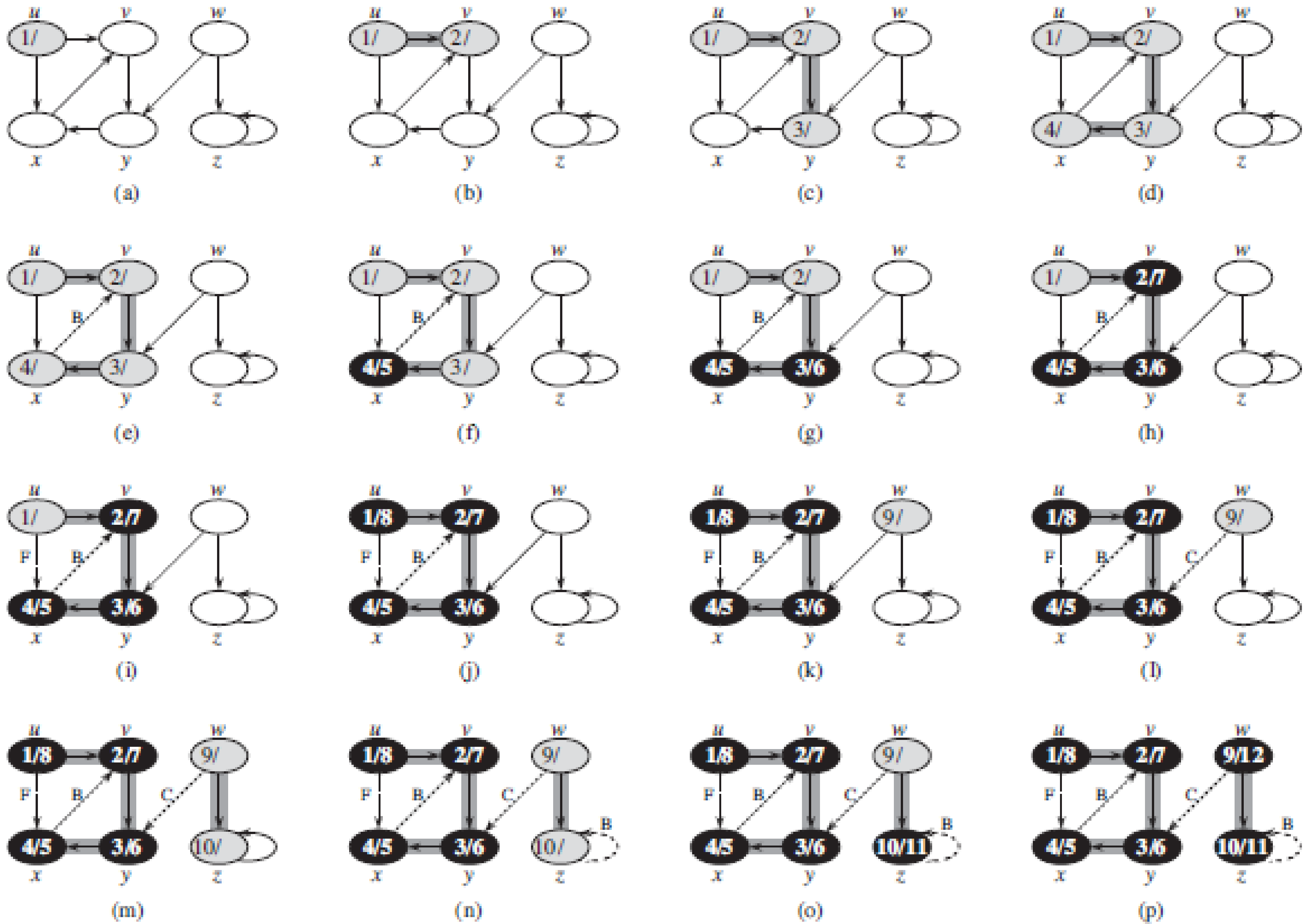
- Uses a global timestamp *time*.

```
DFS_ALL(G)
    for each u ∈ V
        do color[u] ← WHITE
time ← 0
for each u ∈ V
        do if color[u] = WHITE then DFS-VISIT(u)

DFS-VISIT(G,u)
color[u] ← GRAY // discover u
time ← time+1
d[u] ← time
for each v ∈ Adj[u] // explore (u, v)
    do if color[v] = WHITE then DFS-VISIT(v)
color[u] ← BLACK    //
time ← time+1
f[u] ← time              //finish u
```

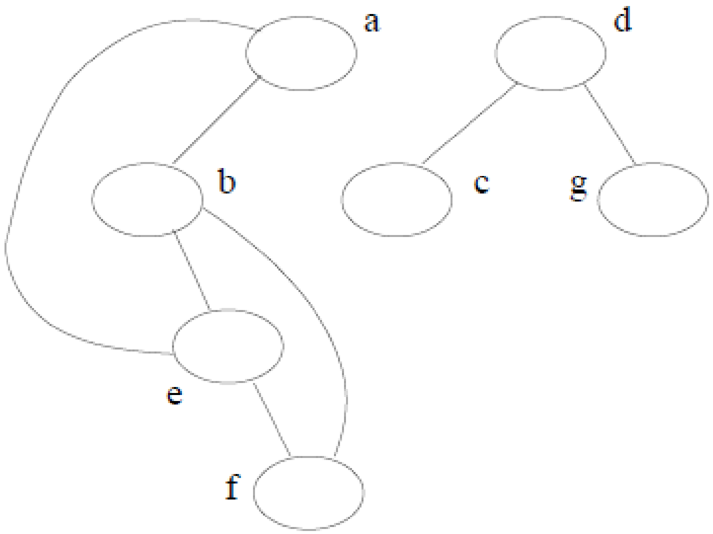
Edge Classification

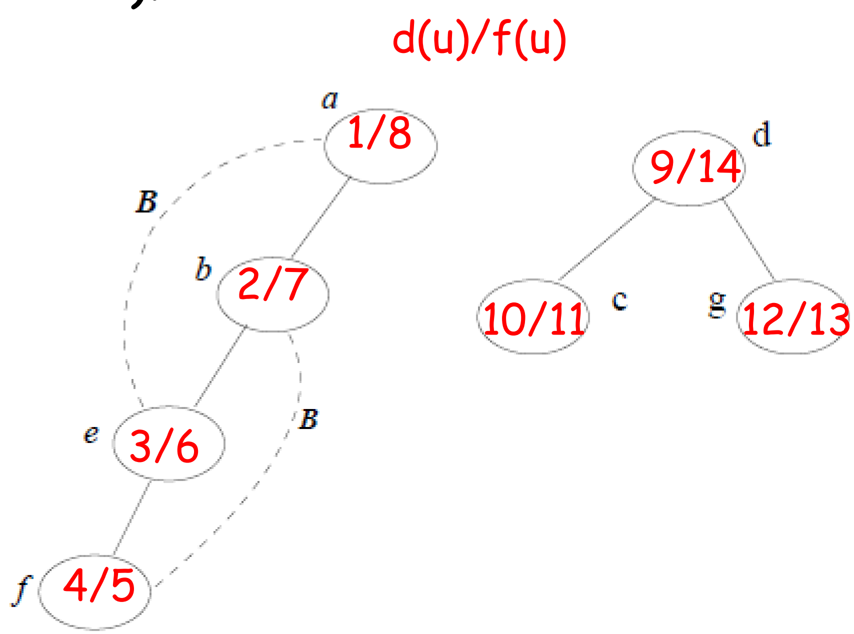
- If we perform a DFS on a graph we can classify the edges of a graph:
  - `Tree edges:` these form part of the search tree (or forest);
  - `Forward edges`: these lead from a vertex to a descendant;
  - `Backward edges`: these lead from a vertex to an ancestor;
  - `Cross edges`: these are all the edges that are left—they connect unrelated vertices.
- Vertex v is descendant of vertex u in depth first search iff v is discovered during the time which u is gray


(a)
(b)
(c)
(d)
(e)
(f)
(g)
(h)
(i)
(j)
(k)
(l)
(m)
(n)
(o)
(p)

Another Example

- Given a graph $G = (V, E)$, it traverses all vertices of G and constructs a forest (a collection of rooted trees), together with a set of source vertices (the roots).



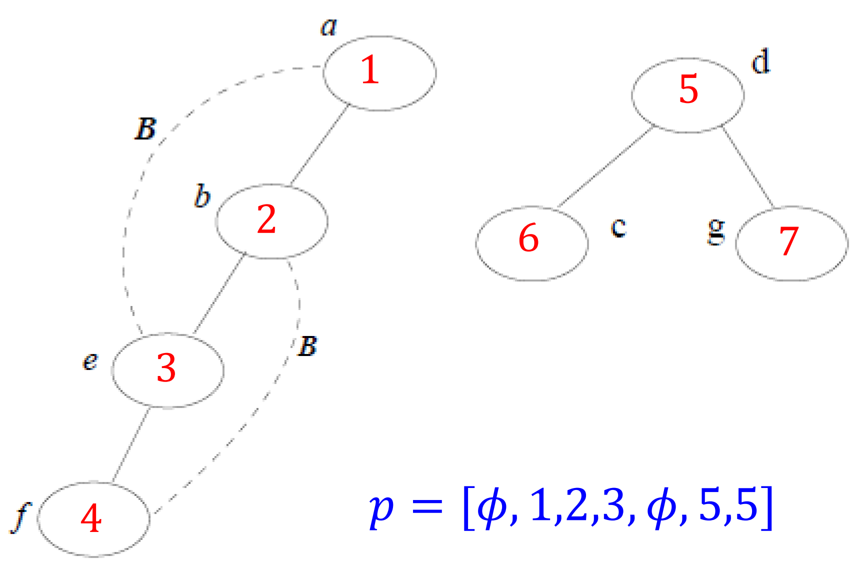


$DSF\_ALL(G)$ - a Simple Version

```
DFS{G} {
   for each v in V do // Initialize
        visit[v] = false;
        p[v] = NULL;
        time=0;
   for each v in V do
        if (visit[v] == false) RDFS(v);
}
RDFS(v) {
   visit[v]=true;
   d[v] = ++time;
   for each w in Adj(v) do {
        if (visit[w] == false) {
           p[w]=v;
           RDFS(w);
        }
   }
}
```

- $d[u]$ - the discovery time, a counter indicating when vertex u is discovered.
- $p[u]$ – the predecessor of u, which discovered u.

$d(u)$



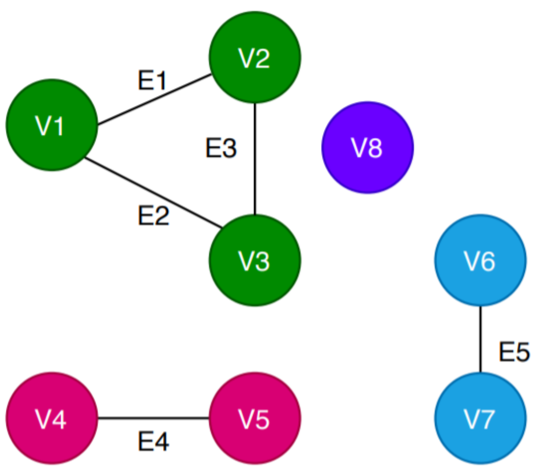
$p = [\phi, 1,2,3, \phi, 5,5]$

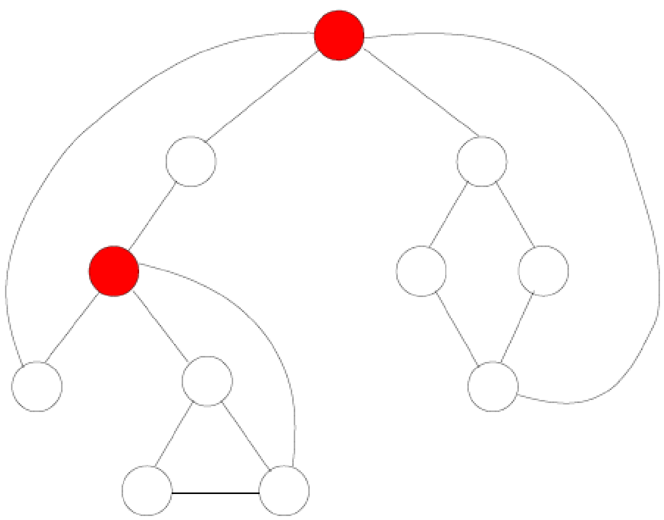
Edge Classification - Undirected Graphs

- For an undirected graph, it can only has tree edges or back edges.
- Tree edges
  - which are the edges $\{p[v], v\}$ where DFS calls are made.
- Back edges
  - which are the edges $\{u, v\}$ where v is an ancestor of u in the DFS tree.

Connected Components in a Graph

- A connected component is **a maximal connected subgraph of an undirected graph**. Each vertex belongs to exactly one connected component, as does each edge.
- A graph is connected if and only if it has exactly one connected component.



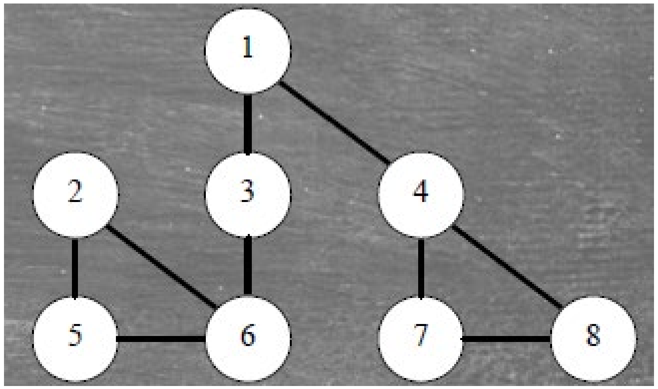


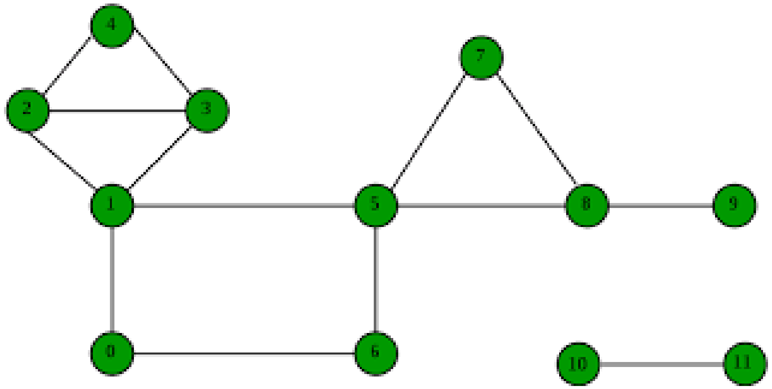
Articulation Points

- A vertex, v, in an undirected connected graph is an articulation point (or cut vertex) if removing it (and edges through it) disconnects the graph.
- Articulation points represent vulnerabilities in a connected network – single points whose failure would split the network into 2 or more disconnected components.
- They are useful for designing reliable networks.
- For a $disconnected$ undirected graph, an articulation point is a vertex removing which increases number of connected components.

Articulation Points

- Which vertices are articulation points?





- How can we find articulation points in a systematic manner?

A Naïve Approach

- A simple approach is to one by one remove all vertices and see if removal of a vertex causes disconnected graph. Following are steps of simple approach for connected graph.

- For every vertex v, do following
  - Remove v from graph.
  - See if the graph remains connected (We can either use BFS or DFS)
  - Add v back to the graph.

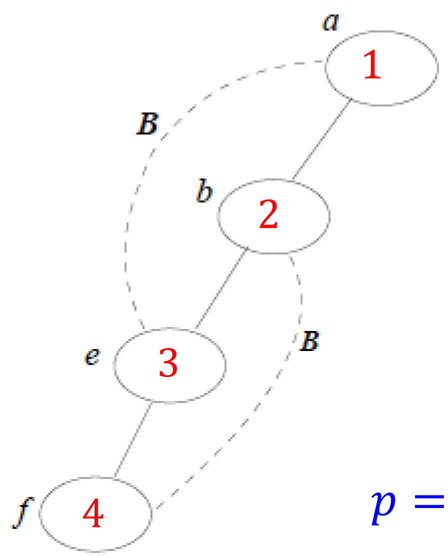
- Time complexity $O\big(V * (V + E)\big)$
- Can we do better?

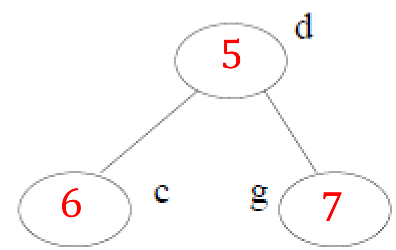
Observations

1. The root of the DFS tree is an articulation if it has two or more children.

2. Any other internal vertex v in the DFS tree, if it has a subtree rooted at a child of v that does **NOT** have an edge which climbs 'higher' than v, then v is an articulation point.

$d(u)$

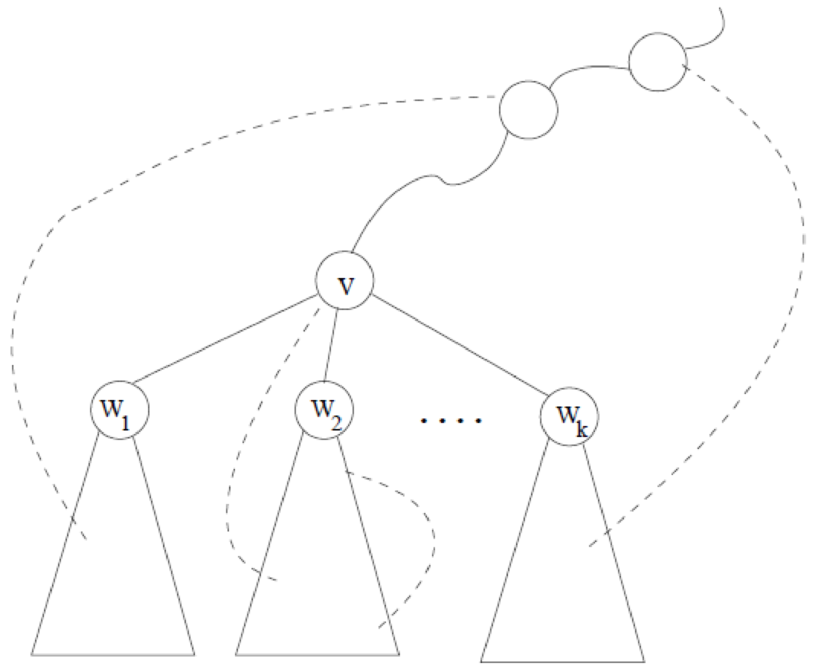




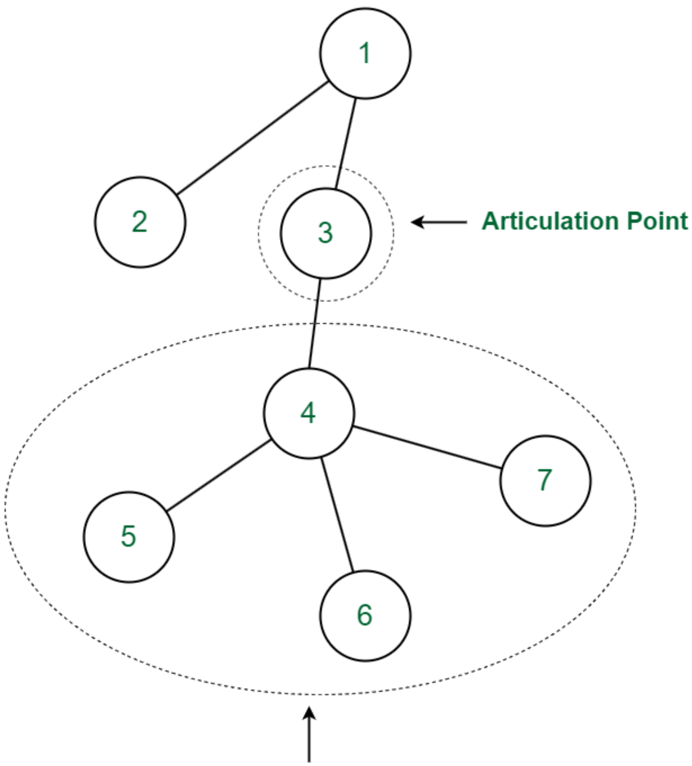
$p = [\phi, 1,2,3, \phi, 5,7]$

How to climb up

- For an undirected graph, it can only has tree edges or back edges. A subtree can only climb to the upper part of the tree by a back edge, and a vertex can only climb up to its ancestor.

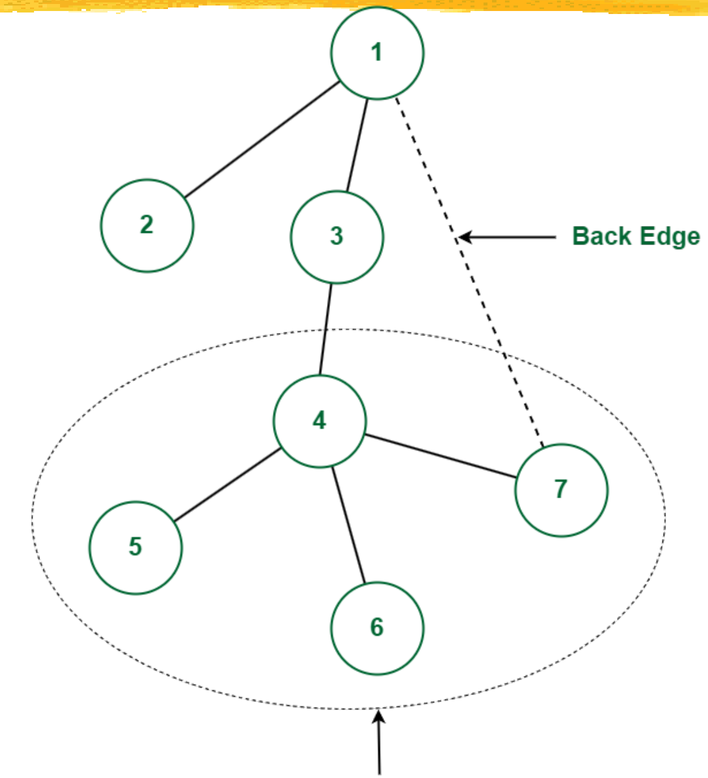


How to climb up



For the subtree rooted at vertex 4, none of the vertices in this subtree has a back edge to one of the ancestors of vertex 3.

So this subtree will get disconnected on the removal of vertex 3.



For the subtree rooted at vertex 4, vertex 7 in this subtree has a back edge to one of the ancestors of vertex 3.

So this subtree will not get disconnected on the removal of vertex 3.

Observation 2

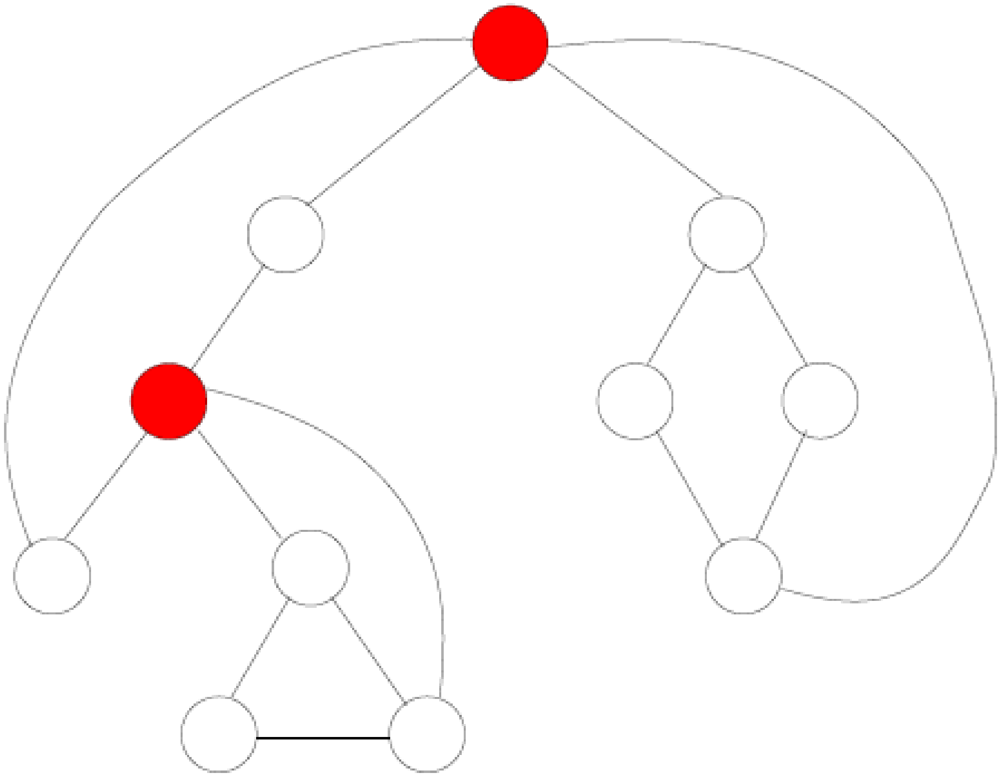
- We make use of the *discovery time* in the DFS tree to define 'low' and '$high$'. Observe that if we follow a path from an ancestor ($high$) to a descendant (low), the $discovery\ time$ is in $increasing\ order$.
- If there is a subtree rooted at a children of v which does not have a back edge connecting to a $SMALLER$ $discovery\ time$ than $discover[v]$, then v is an articulation point.
- How do we know a subtree has a back edge climbing to an upper part of the tree ?

Observation 2…

- Define $Low(v)$ to denote the earliest (highest) visited vertex (the vertex with minimum discovery time) that can be reached from the subtree rooted with v
- $Low(v) =$ $\min\{discover(v);\ discover(w): (u, w)$ is a back edge for some descendant u of $v\}$
- w is an ancestor of v
- u is a descendant of v

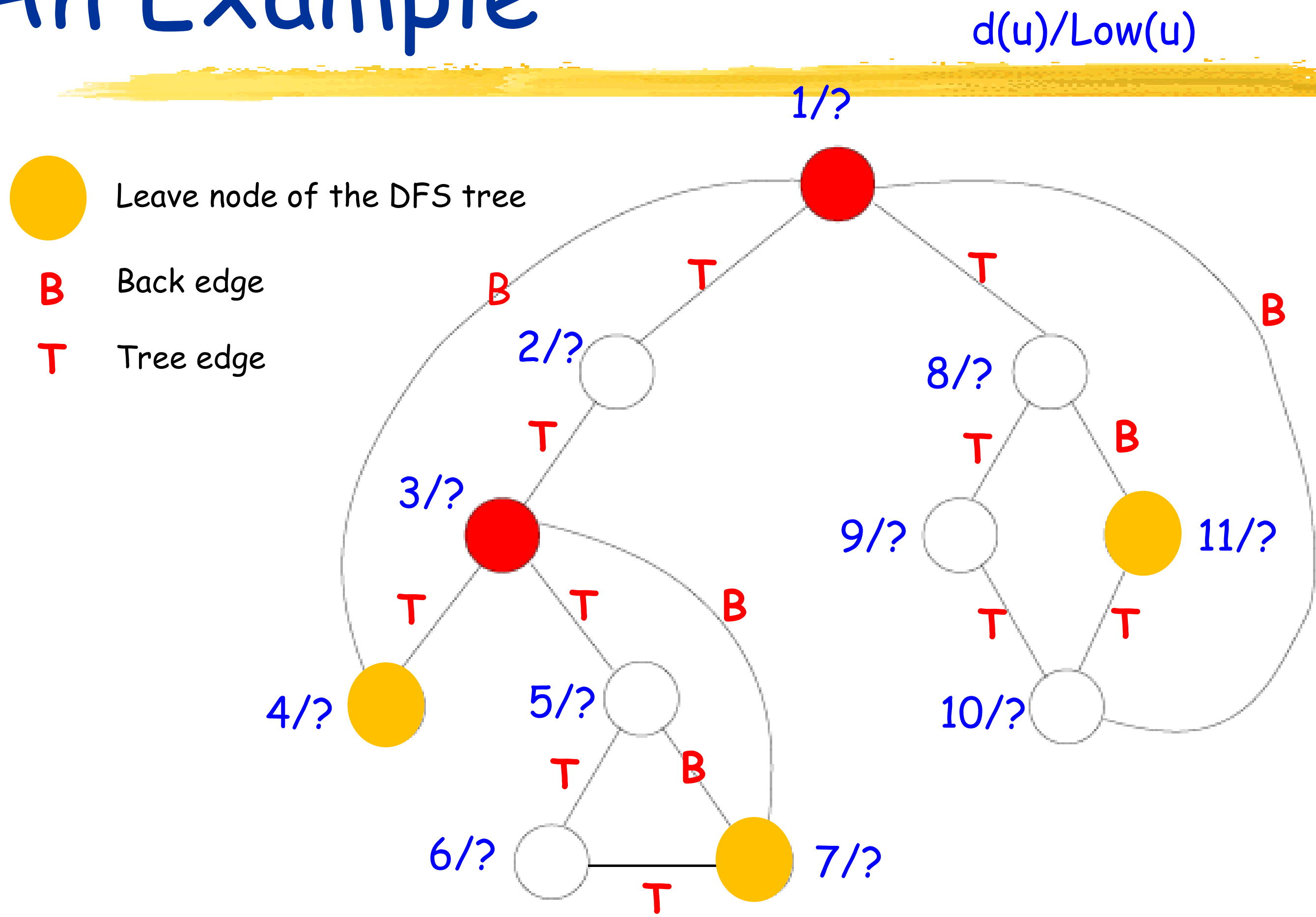
An Example

d(v)/Low(v)



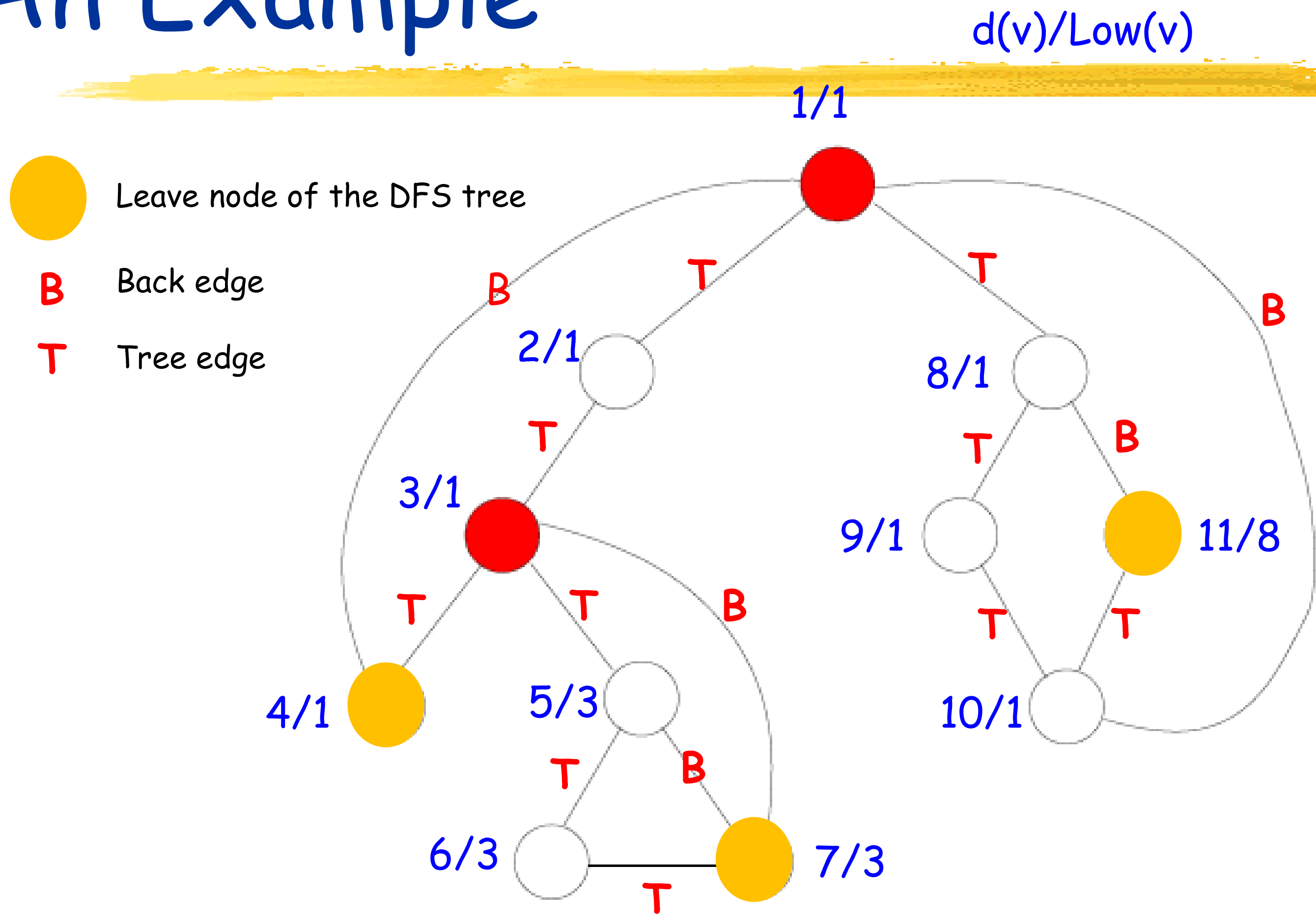
$Low(v) = \min\{discover(v); discover(w) : (u, w)$ **is a back edge for some descendant** u **of** $v\}$

An Example



$Low[v] = \min\{discover[v]; discover[w] : (u; w)$ **is a back edge for some descendant** u **of** $v\}$

An Example



$Low(v) = \min\{discover(v);\ discover(w):\ (u,w)$ **is a back edge for some descendant** u **of** $v\}$

Compute $Low(v)$

```
RDFS_Compute_Low(v) {
   visit[v]=true;
   Low[v]=discover[v] = ++time;
   for each w in Adj(v) do
        if (visit[w] == false) {
           p[w]=v;
           RDFS_Compute_Low(w);
           // When RDFS_Compute_Low(w) returns,
           // Low[w] stores the
           // lowest value it can climb up
           // for a subtree rooted at w.
           Low[v] = min(Low[v], Low[w]);
        } else if (w != p[v]) {
           // {v, w} is a back edge
           Low[v] = min(Low[v], discover[w]);
   }
}
```

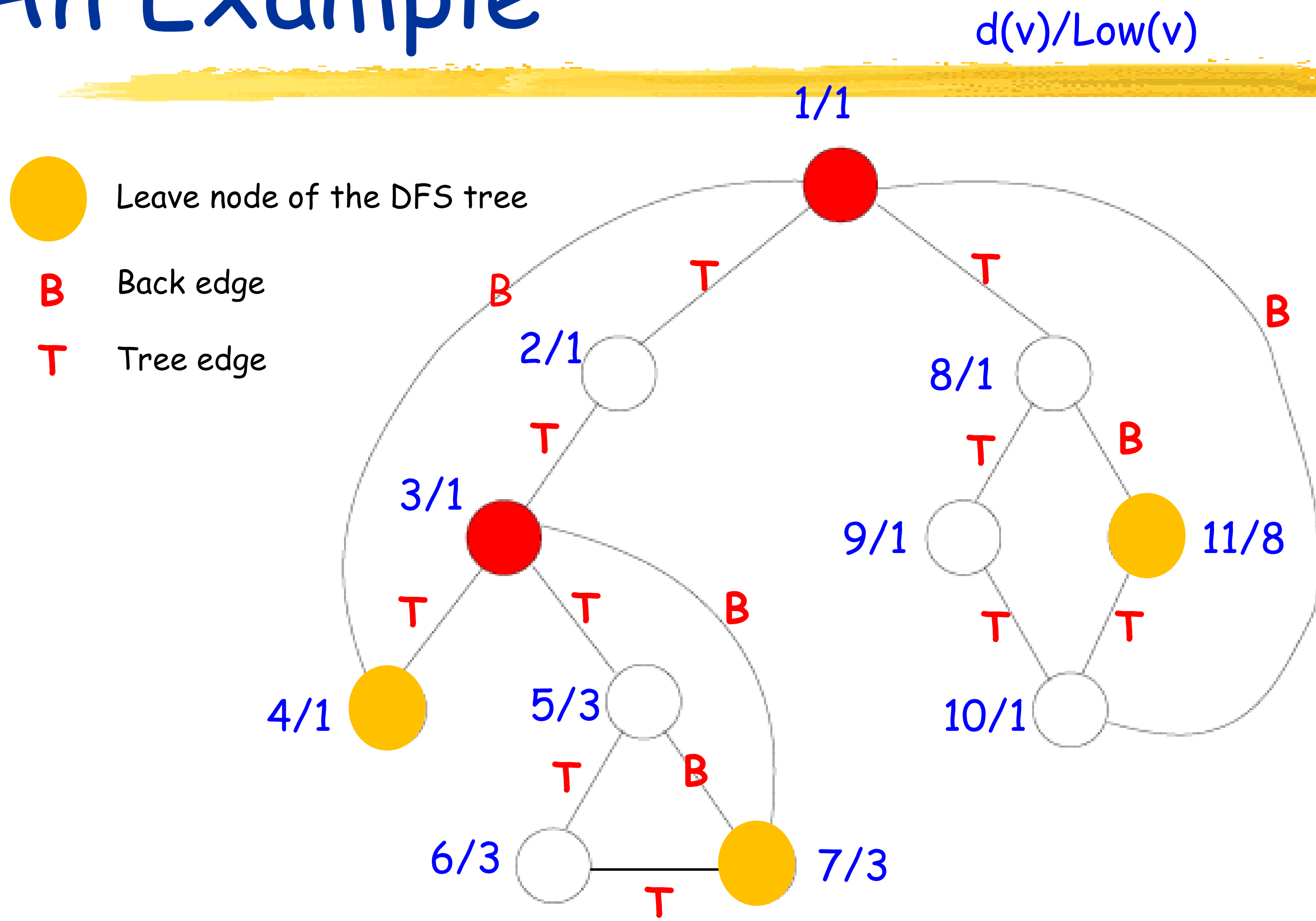
Articulation Points

- Conduct a depth first traversal of G, producing the spanning tree T with $discovery[v]$ and $Low[v]$ for each node v
- Consider each node v of the graph:
  - The root of the DFS tree, T, is an articulation point if it has two or more children.
  - If v has no children in T, it is not an articulation point;
  - Any other $internal$ (non-leaf) vertex v in the DFS tree is an articulation point if v has a *non-leave child* w such that $Low[w] \geq discover[v]$.

Articulation Points…

```
// search for the DFS tree T
ArticulationPoints {
   for each v in V do {
        articulation_point(v) = false;
        if (p[v] == NULL) { //v is a root
           if (|Adj(v)|>1)
               articulation_point(v)=true;
           } else{
               for each w in Adj(v) and w is not a leaf in T do {
                  if (Low[w] >= discover[v])
                      articulation_point(v)=true;
           }
        }
   }
}
```

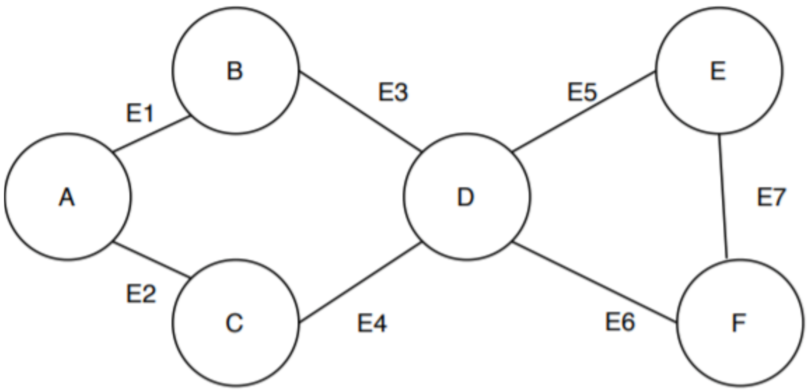
An Example



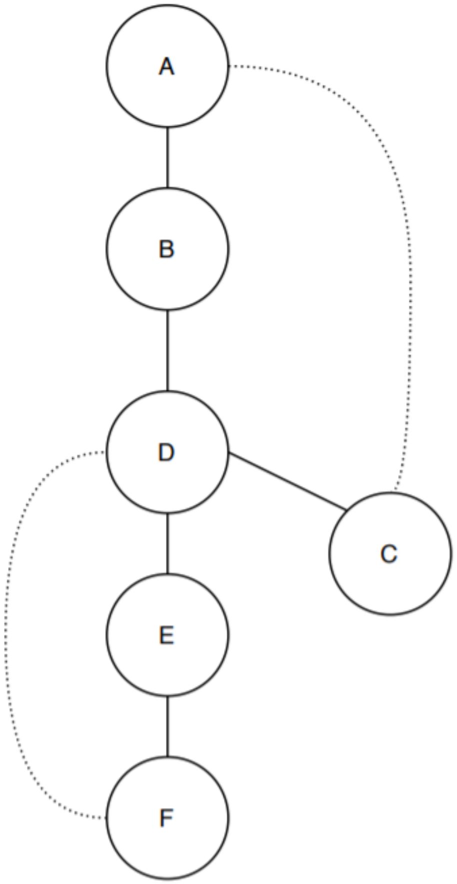
$Low(v) = \min\{discover(v);\ discover(w)$: (u, w) **is a back edge for some descendant** u **of** $v\}$

Another Example

We will run the algorithm to find out the articulation points:



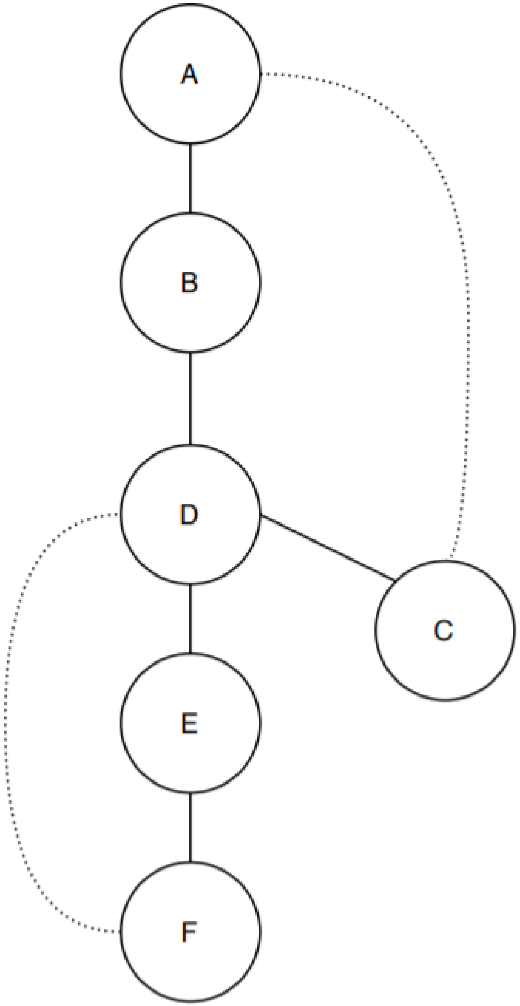
First, we convert the graph to a DFS tree (take Vertex A as the root).



Another Example

Second, we calculate the lowest discovery number of each vertex

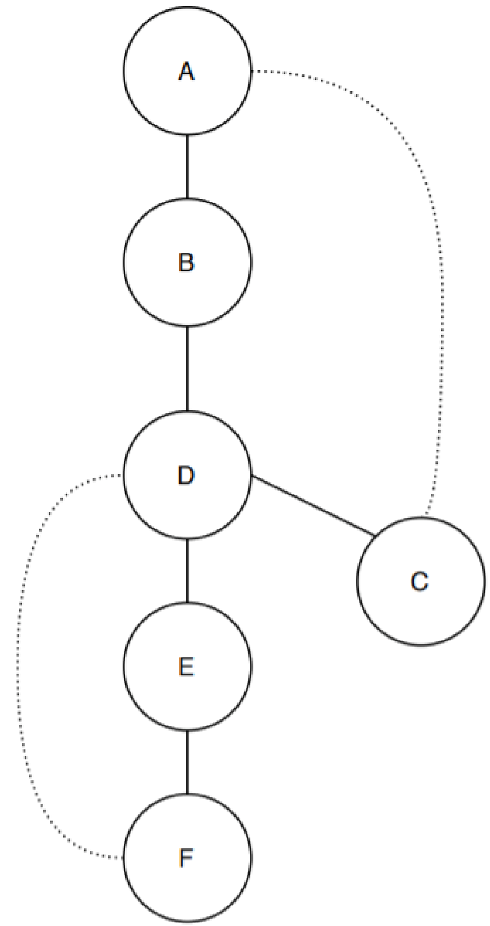
Vertex	Depth Number	Lowest Discovery Number
A	1	1
B	2	1
C	6	1
D	3	1
E	4	3
F	5	3



Another Example

Third, find the articulation points. Let's start with a random pair of vertices (E, F) where E is the parent of F. Checking condition: low[F] >= depth[E] ? but 3 < 4, so E is not an articulation point.

Now let's take another pair (D, E) where D is the parent of E. Checking condition: low[E] >= depth[D]? and 3=3 Checking condition D is not a leaf, so D is an articulation point.



Vertex	Depth Number	Lowest Discovery Number
A	1	1
B	2	1
C	6	1
D	3	1
E	4	3
F	5	3

Why?

- The identification of articulation points is important in determining the critical components of networks.
- A component which corresponds to an articulation point is critical.
- If such a component fails, the network is compromised.