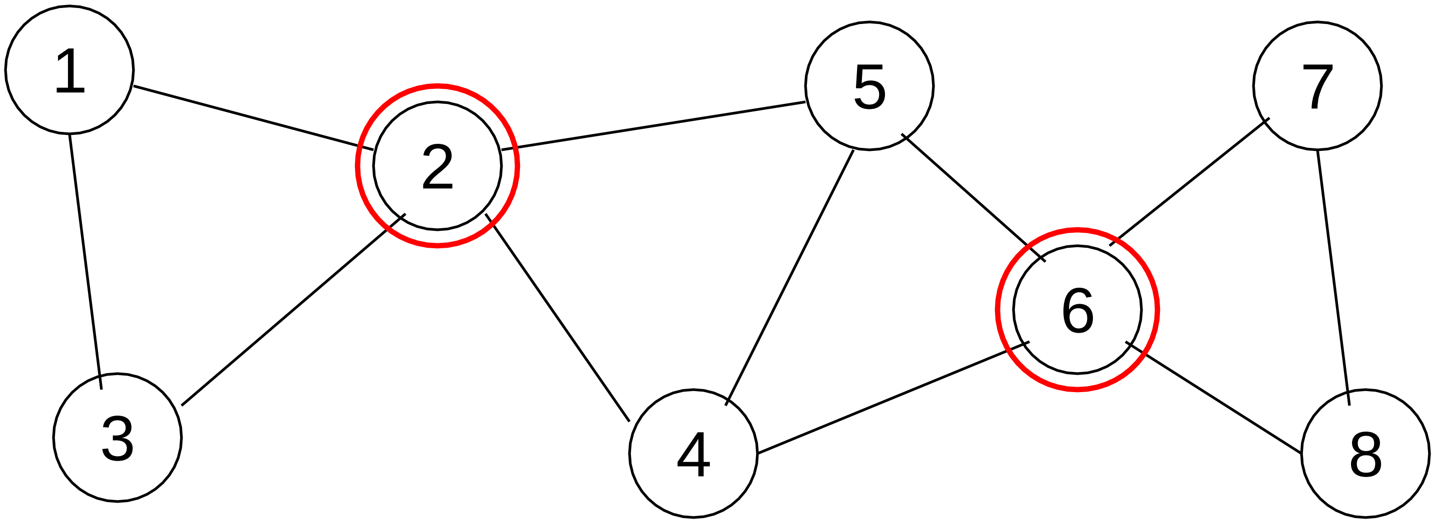
# Biconnected components

# Articulation points, Bridges, Biconnected Components

- Let G = (V;E) be a connected, undirected graph.
- An ***articulation point / separation vertex*** of G is a vertex whose removal disconnects G.
- A ***bridge / separation edge*** of G is an edge whose removal disconnects G
- **A connected graph G is biconnected** if, for any two vertices u and v of G, there are two disjoint paths between u and v, that is, two paths sharing no common edges or vertices, except u and v**.**

- These concepts are important because they can be used to identify vulnerabilities of networks

# Articulation points – Example



# Articulation points – Example

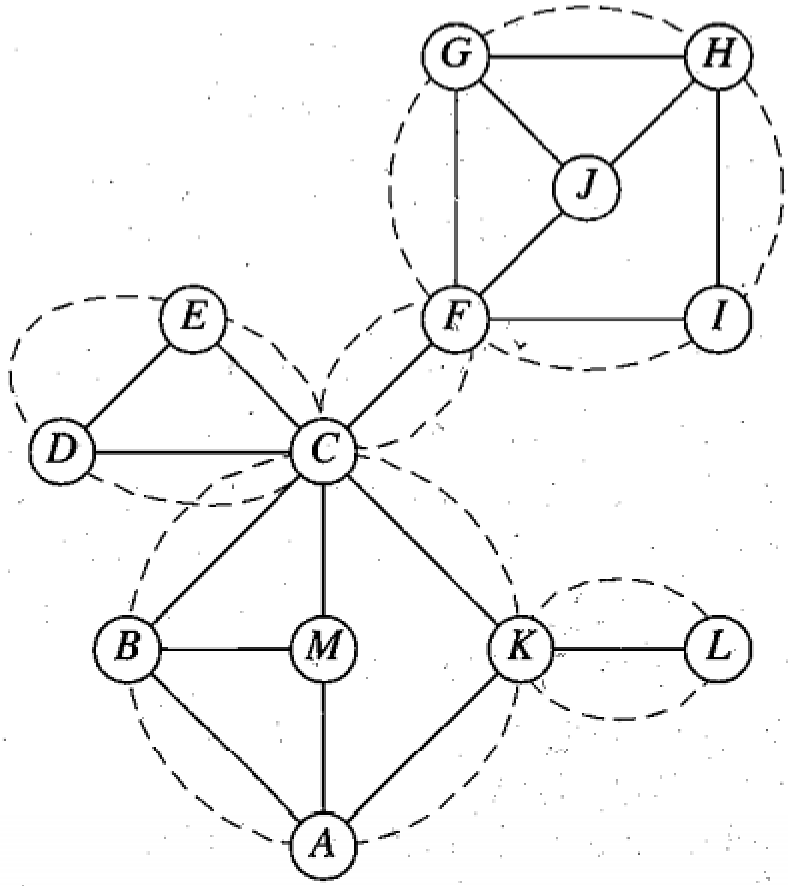



**Figure 6.8:** Biconnected components, shown circled with dashed lines. $C$, $F$, and $K$ are separation vertices; $(C, F)$ and $(K, L)$ are separation edges.

# How to find all articulation points ?

- **Brute-force approach**: one by one remove all vertices and see if removal of a vertex causes the graph to disconnect:

  For every vertex v, do :

    Remove v from graph

    See if the graph remains connected (use BFS or DFS)

    If graph is disconnected, add v to AP list

    Add v back to the graph

- Time complexity of above method is O(n*(n+m)) for a graph represented using adjacency list.

# How to find all articulation points ?

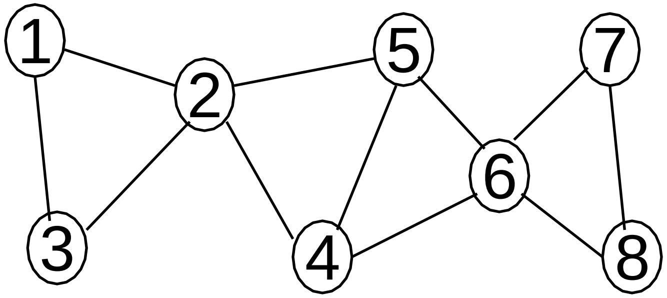
- **DFS- based-approach**:
- We can prove following properties:

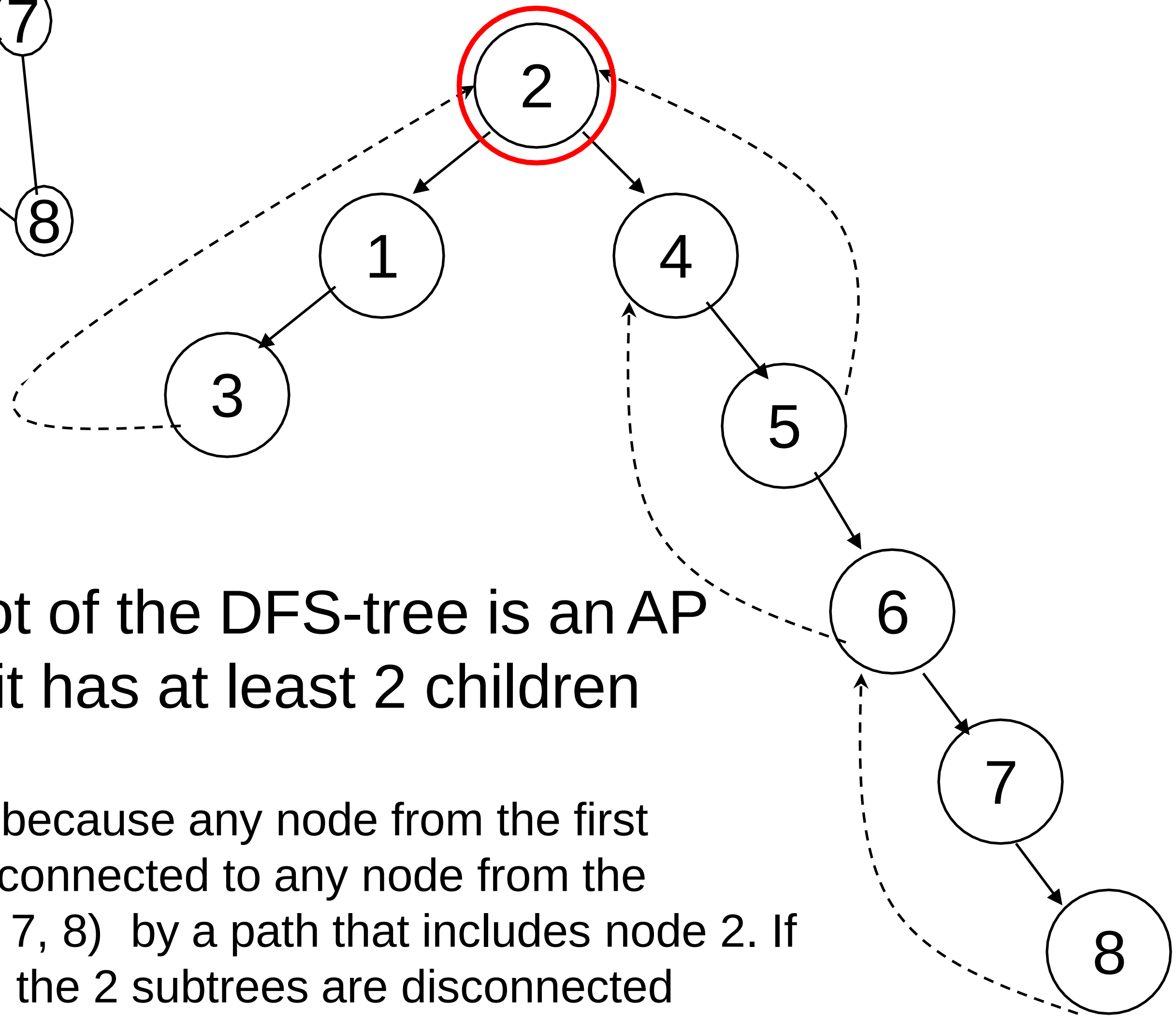
1. The root of a DFS-tree is an articulation point if and only if it has at least two children.

2. A nonroot vertex v of a DFS-tree is an articulation point of G if and only if has a **child s** such that there is **no back edge from s or any descendant** of s to a **proper ancestor of v**.

3. Leafs of a DFS-tree are never articulation points

# Finding articulation points by DFS

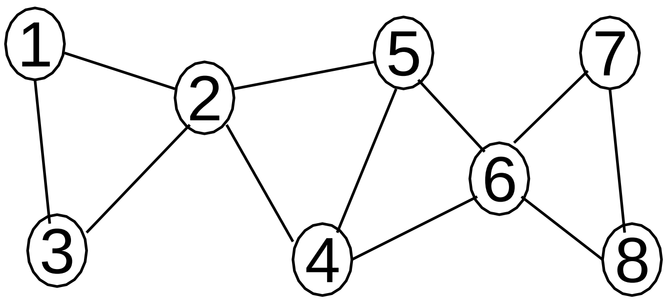


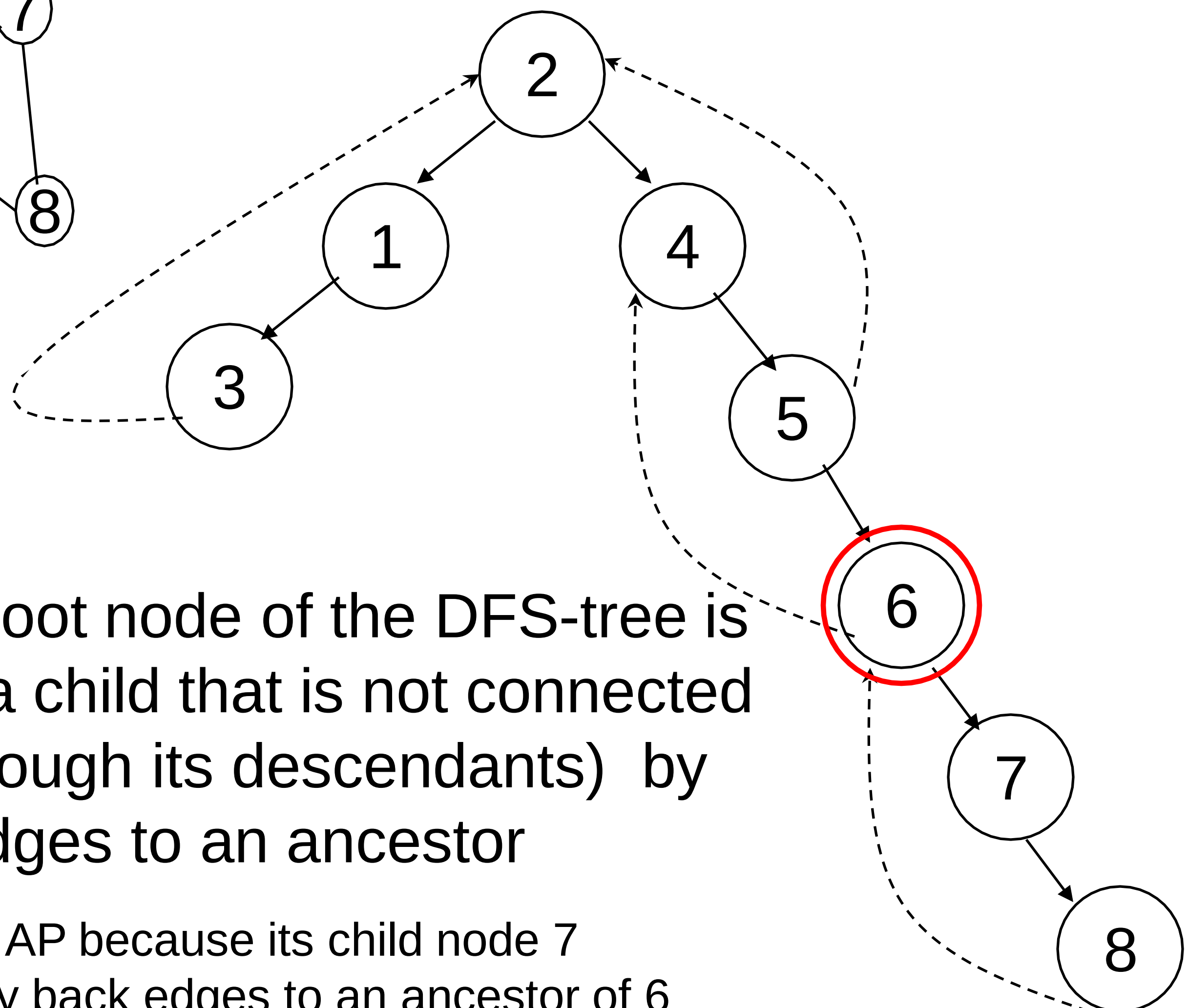




Case 1: The root of the DFS-tree is an AP if and only if it has at least 2 children

Node 2 is an AP because any node from the first subtree (1, 2) is connected to any node from the second subtree (4, 5, 6, 7, 8) by a path that includes node 2. If node 2 is removed, the 2 subtrees are disconnected

# Finding articulation points by DFS

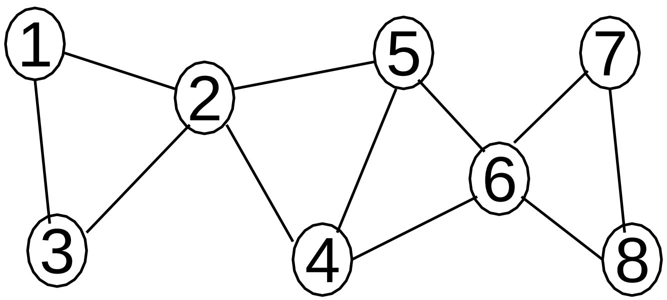


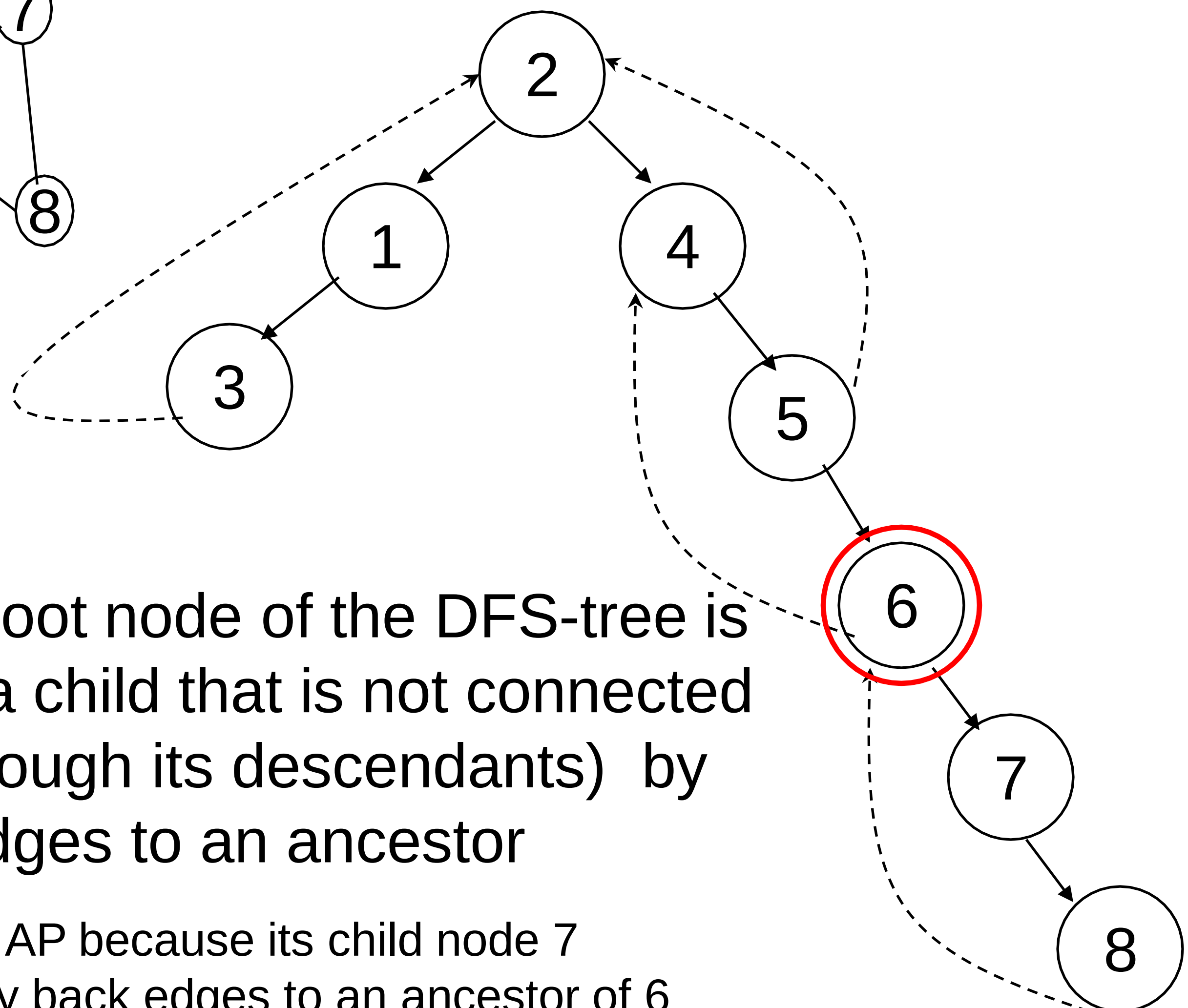




Case 2: A non-root node of the DFS-tree is an AP if it has a child that is not connected (directly or through its descendants) by back edges to an ancestor

Node 6 is an AP because its child node 7
is not connected by back edges to an ancestor of 6

# Finding articulation points by DFS




Case 2: A non-root node of the DFS-tree is an AP if it has a child that is not connected (directly or through its descendants) by back edges to an ancestor

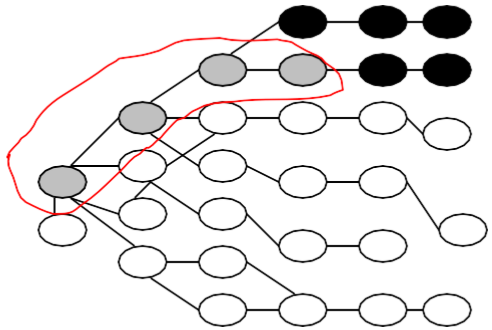
Node 6 is an AP because its child node 7 is not connected by back edges to an ancestor of 6

# Depth-First Search: Revisited

- *Depth-first search* is another strategy for exploring a graph
  - Explore “deeper” in the graph whenever possible
  - Edges are explored out of the most recently discovered vertex *v* that still has unexplored edges
  - When all of *v*’s edges have been explored, backtrack to the vertex from which *v* was discovered

# Depth-First Search

- Vertices initially colored white (flag=-1)
- Then colored gray when discovered (flag=0)
- Then black when their exploration is finished (flag=1)



# Depth-First Search

- Every vertex v will get following attributes:
  - v.color: (white, grey, black) – represents its exploration status
  - v.pi represents the “parent” node of v (v has ben reached as a result of exploring adjacencies of pi)
  - v.d represents the time when the node is discovered
  - v.f represents the finishing time when the node

DFS($G$)

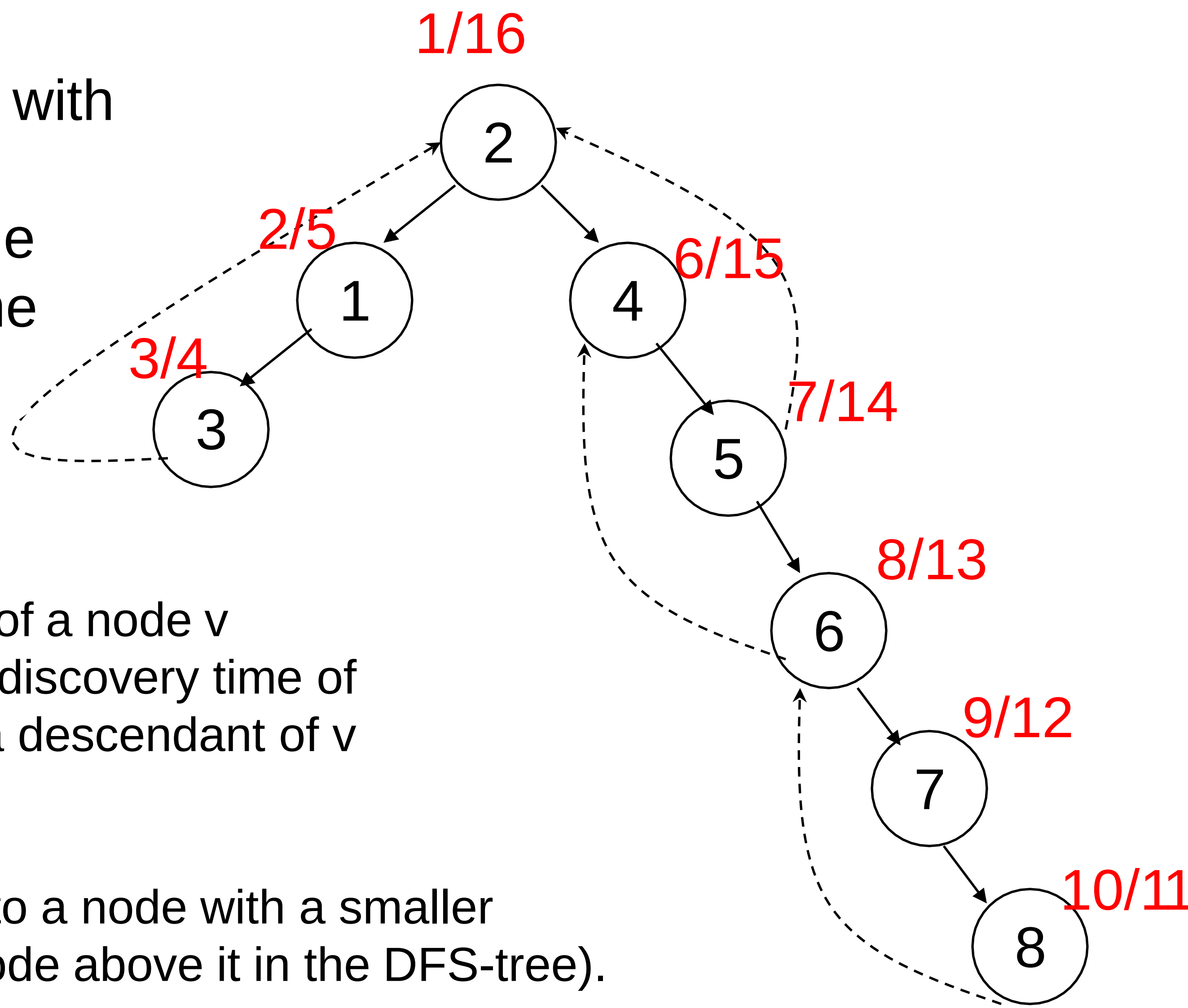
```
for each vertex u ∈ G.V
    u.color = WHITE
    u.π = NIL
time = 0
for each vertex u ∈ G.V
    if u.color == WHITE
        DFS-VISIT(G, u)
```

DFS-VISIT($G, u$)

```
time = time + 1
u.d = time
u.color = GRAY
for each v ∈ G.Adj[u]
    if v.color == WHITE
        v.π = u
        DFS-VISIT(G, v)
u.color = BLACK
time = time + 1
u.f = time
```

# Reminder: DFS – v.d and v.f

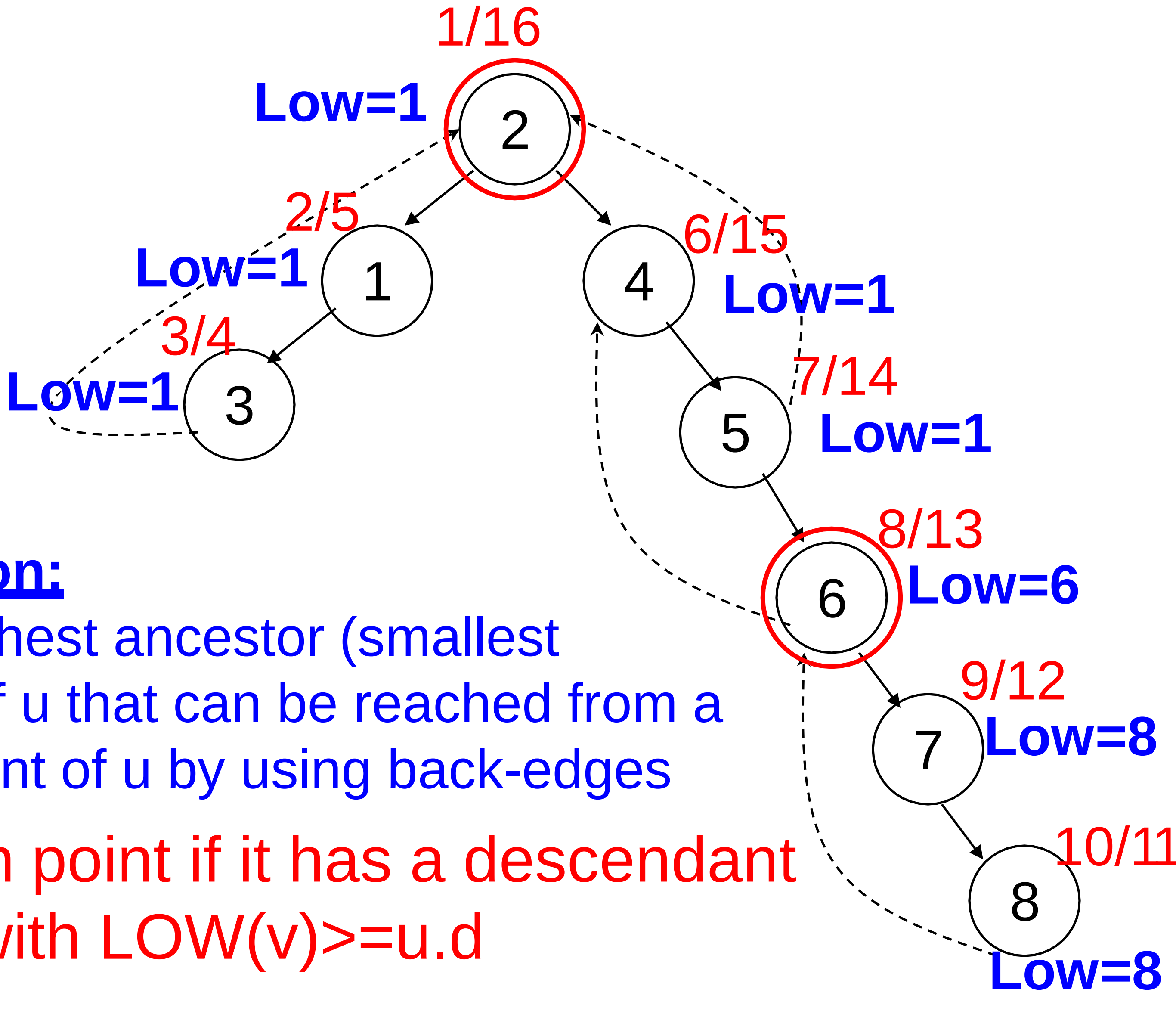
DFS associates with
every vertex v
its discovery time
and its finish time
v.d /v.f




The discovery time of a node v
is smaller than the discovery time of
any node which is a descendant of v
in the DFS-tree.

A back-edge leads to a node with a smaller
discovery time (a node above it in the DFS-tree).

# The LOW function




**<u>The LOW function:</u>**

LOW(u) = the highest ancestor (smallest discovery time) of u that can be reached from a descendant of u by using back-edges

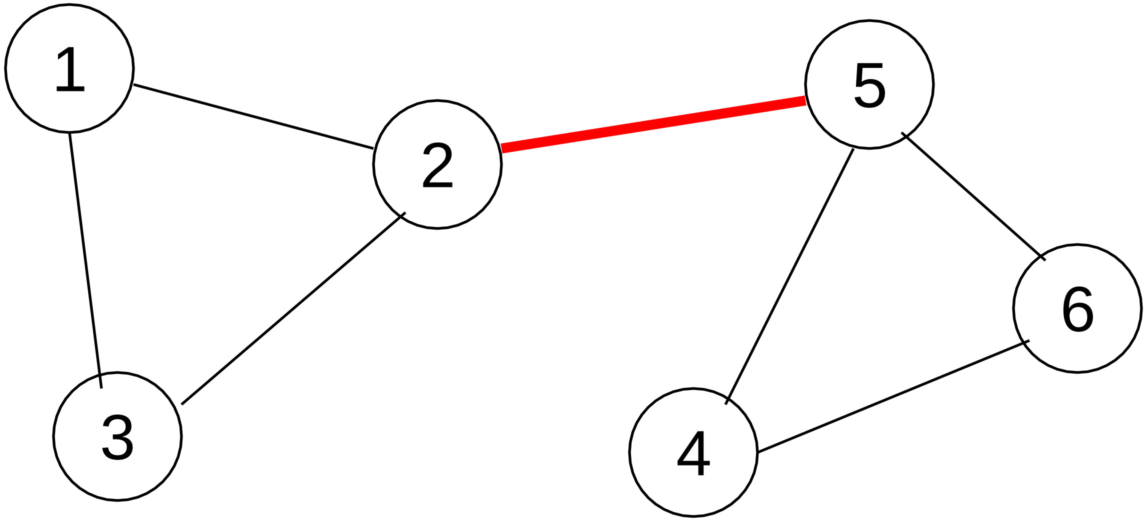
u is articulation point if it has a descendant v with LOW(v)>=u.d

# Finding Articulation Points

- *Algorithm principle:*
  - *During DFS, calculate also the values of the LOW function for every vertex*
  - *After we finish the recursive search from a child v of a vertex u, we update u.low with the value of v.low. Vertex u is an articulation point, disconnecting v, if v.low >=u.d*
  - *If vertex u is the root of the DFS tree, check whether v is its second child*
  - *When encountering a back-edge (u,v) update u.low with the value of v.d*

```
DFS_VISIT_AP(G, u)
      time=time+1
      u.d=time
      u.color=GRAY
      u.low=u.d
      for each v in G.Adj[u]
            if v.color==WHITE
                  v.pi=u
                  DFS_VISIT_AP(G,v)
                  if (u.pi==NIL)
                        if (v is second son of u)
                              "u is AP"    // Case 1
                  else
                        u.low=min(u.low, v.low)
                        if (v.low>=u.d)
                               "u is AP"    // Case 2
            else  if ((v<>u.pi) and (v.d <u.d))
                        u.low=min(u.low, v.d)
      u.color=BLACK
      time=time+1
      u.f=time
```

# Bridge edges – Example



# How to find all bridges ?

- **Brute-force approach**: one by one remove all edges and see if removal of an edge causes the graph to disconnect:

  For every edge e, do :

    Remove e from graph

    See if the graph remains connected (use BFS or DFS)

    If graph is disconnected, add e to B list

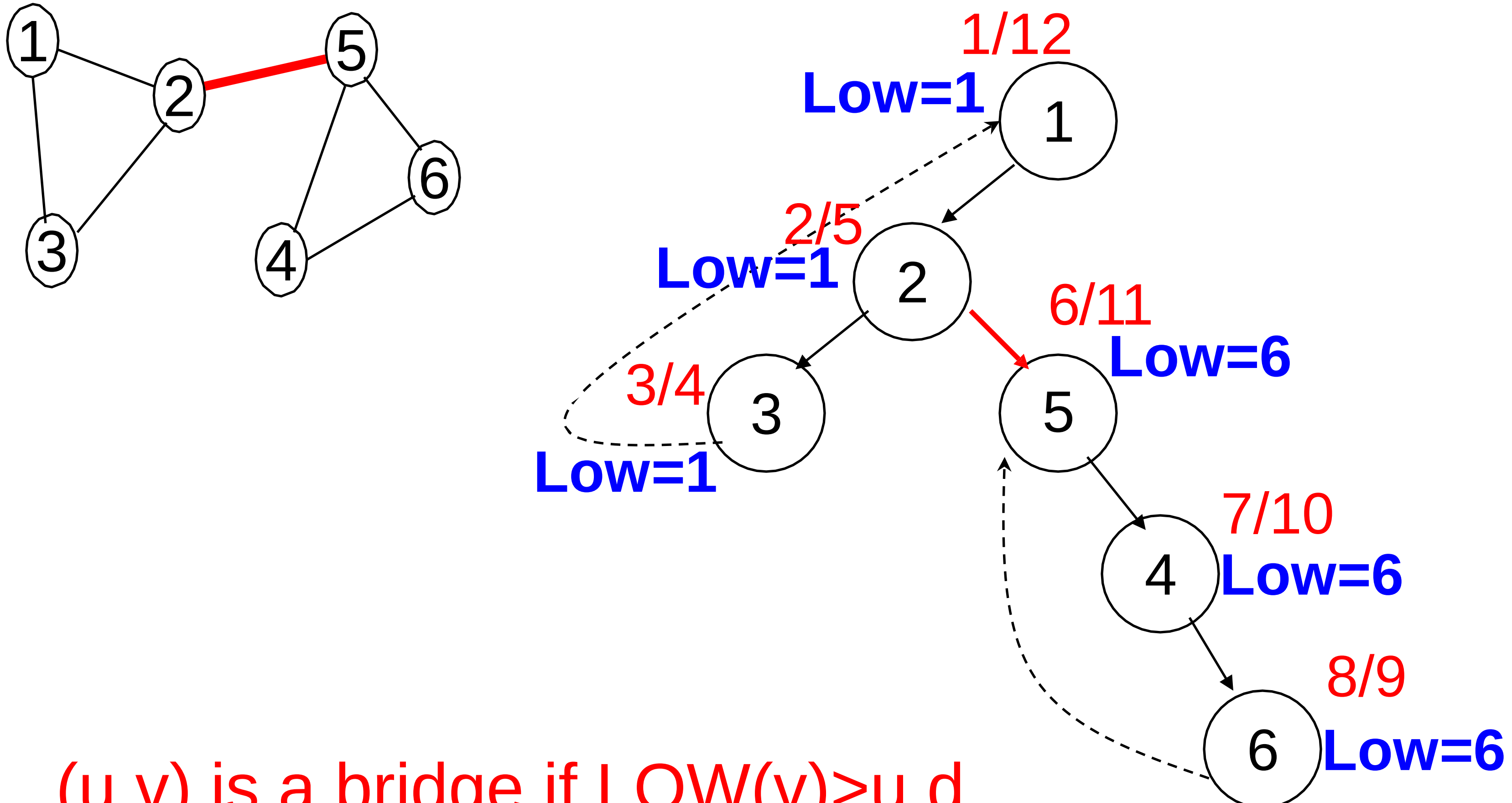
    Add e back to the graph

- Time complexity of above method is O(m*(n+m)) for a graph represented using adjacency list.
- Can we do better?

# How to find all bridges ?

- **DFS- approach**:
- An edge of G is a bridge if and only if it does not lie on any simple cycle of G.
- if some vertex u has a back edge pointing to it, then no edge below u in the DFS tree can be a bridge. The reason is that each back edge gives us a cycle, and no edge that is a member of a cycle can be a bridge.
- if we have a vertex v
  whose parent in the DFS tree is u, and no ancestor of v has a back edge pointing to it, then (u, v) is a bridge.

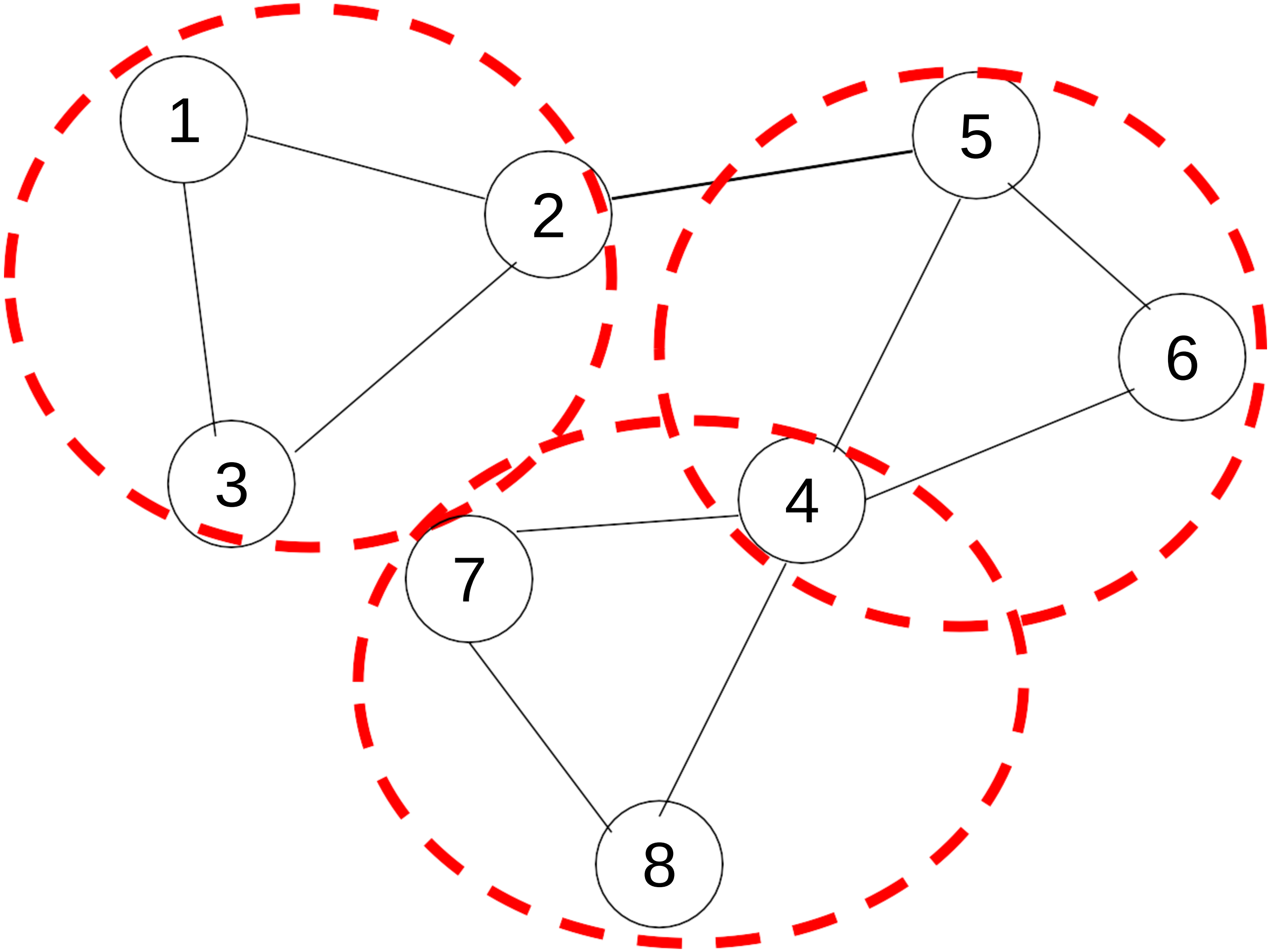
# Finding bridges by DFS




(u,v) is a bridge if LOW(v)>u.d

```
DFS_VISIT_Bridges(G, u)
      time=time+1
      u.d=time
      u.color=GRAY
      u.low=u.d
      for each v in G.Adj[u]
            if v.color==WHITE
                  v.pi=u
                  DFS_VISIT_AP(G,v)

                  u.low=min(u.low, v.low)
                  if (v.low>u.d)
                              "(u,v) is Bridge"
            else  if ((v<>u.pi) and  (v.d <u.d))
                        u.low=min(u.low, v.d)
      u.color=BLACK
      time=time+1
      u.f=time
```

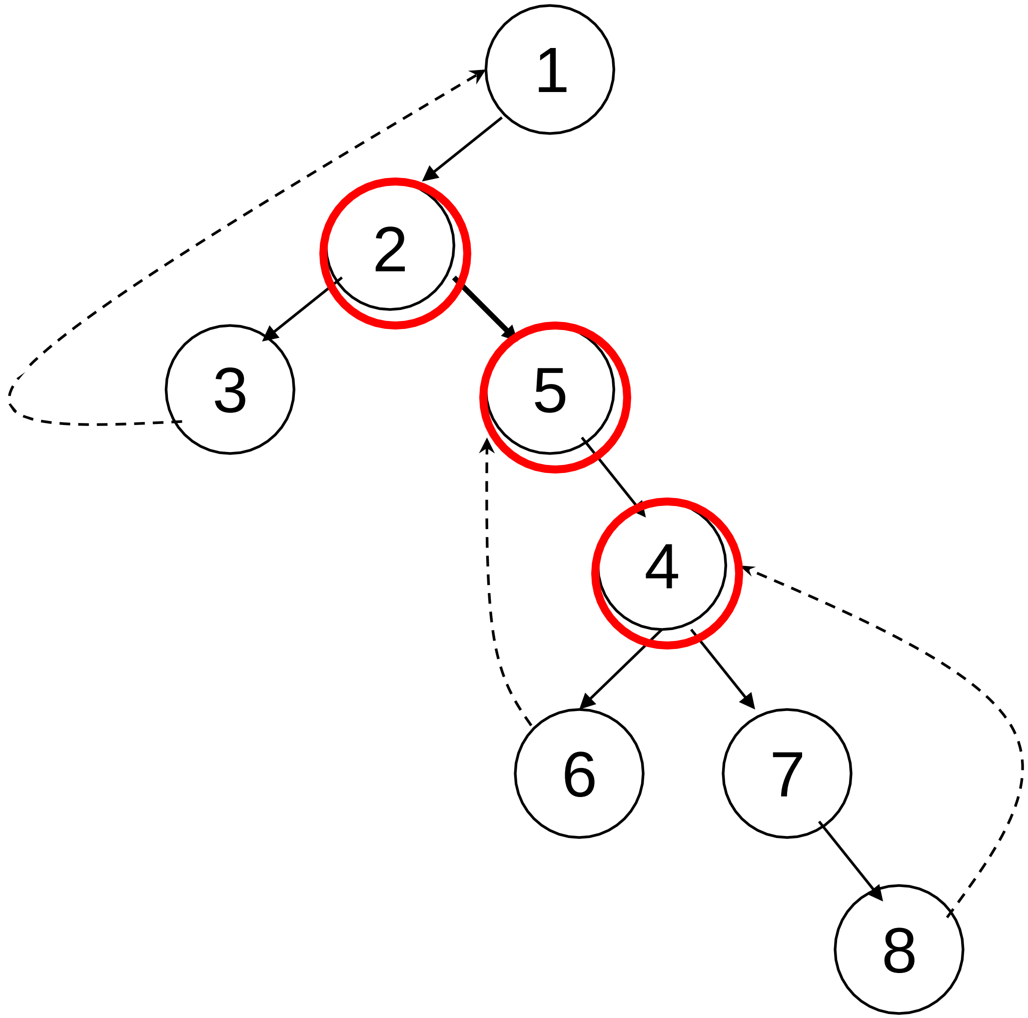
# Biconnected components – Example



# Finding biconnected components

- Two biconnected components cannot have a common edge, but they can have a common vertex
  - -> We will mark the edges with an id of their biconnected component
- The common vertex of several biconnected components is an articulation point
- The articulation points separate the biconnected components of a graph. If the graph has no articulation points, it is biconned
  - -> We will try to identify the biconnected components while searching for articulation points

# Finding biconnected components



# Finding biconnected components

- *Algorithm principle:*
  - *During DFS, use a stack to store visited edges (tree edges or back edges)*
  - *After we finish the recursive search from a child v of a vertex u, we check if u is an articulation point for v. If it is, we output all edges from the stack until (u,v). These edges form a biconnected component*
  - *When we return to the root of the DFS-tree, we have to output the edges even if the root is no articulation point (graph may be biconnex) – we will not test the case of the root being an articulation point*

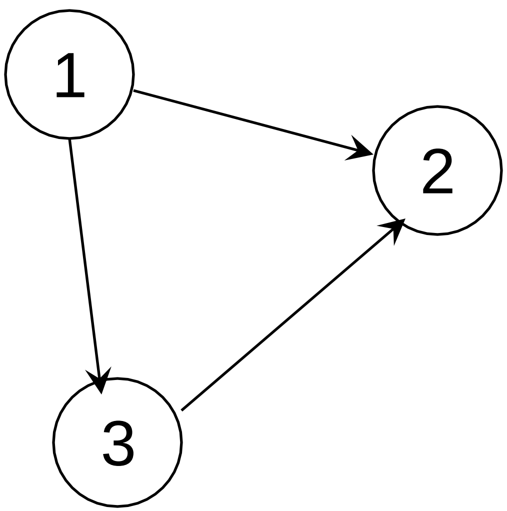
```
DFS_VISIT_BiconnectedComp(G, u)
        time=time+1
        u.d=time
        u.color=GRAY
        u.low=u.d
        u.AP=false
        for each v in G.Adj[u]
                if v.color==WHITE
                        v.pi=u
                        EdgeStack.push(u,v)
                        DFS_VISIT_AP(G,v)
                        u.low=min(u.low, v.low)
                        if (v.low>=u.d)
                                pop all edges from EdgeStack until (u,v)
                                these are the edges of a Biconn Comp
                else    if ((v<>u.pi) and (v.d <u.d))
                                EdgeStack.push(u,v)
                                u.low=min(u.low, v.d)

        u.color=BLACK
        time=time+1
        u.f=time
```

# Applications of DFS

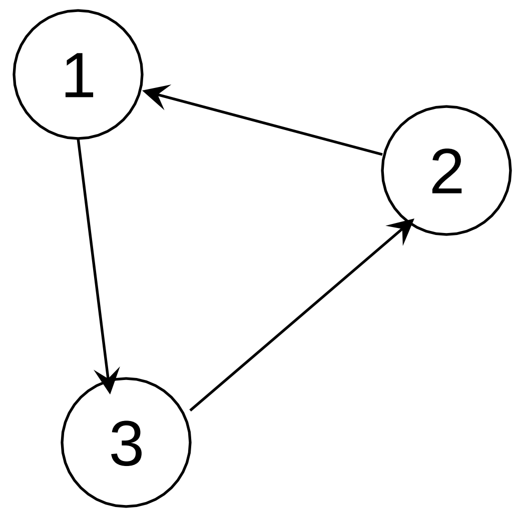
- DFS has many applications
- For undirected graphs:
  - Connected components
  - Connectivity properties
- For directed graphs:
  - Finding cycles
  - Topological sorting
  - Connectivity properties: Strongly connected components

# Directed Acyclic Graphs

- A *directed acyclic graph* or *DAG* is a directed graph with no directed cycles



acyclic

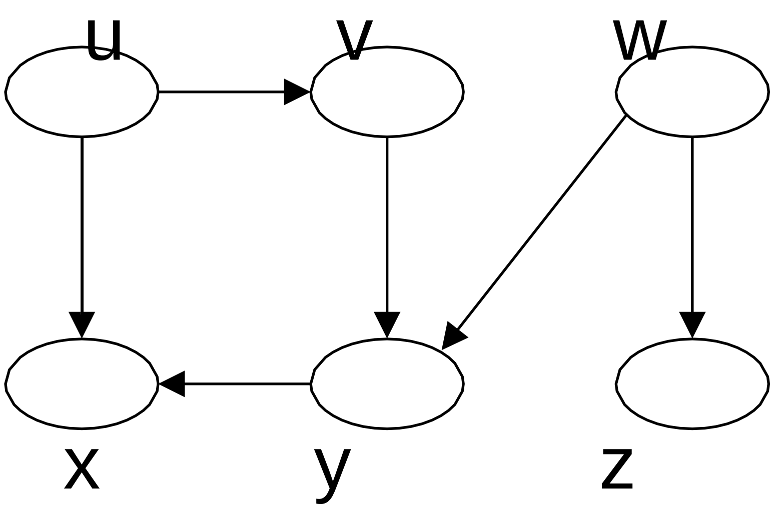


cyclic

# Topological Sort

- *Topological sort* of a DAG (Directed Acyclic Graph):
  - Linear ordering of all vertices in a DAG G such that vertex $u$ comes before vertex $v$ if there is an edge $(u, v) \in G$
  - This property is important for a class of *scheduling* problems

# Example – Topological Sorting



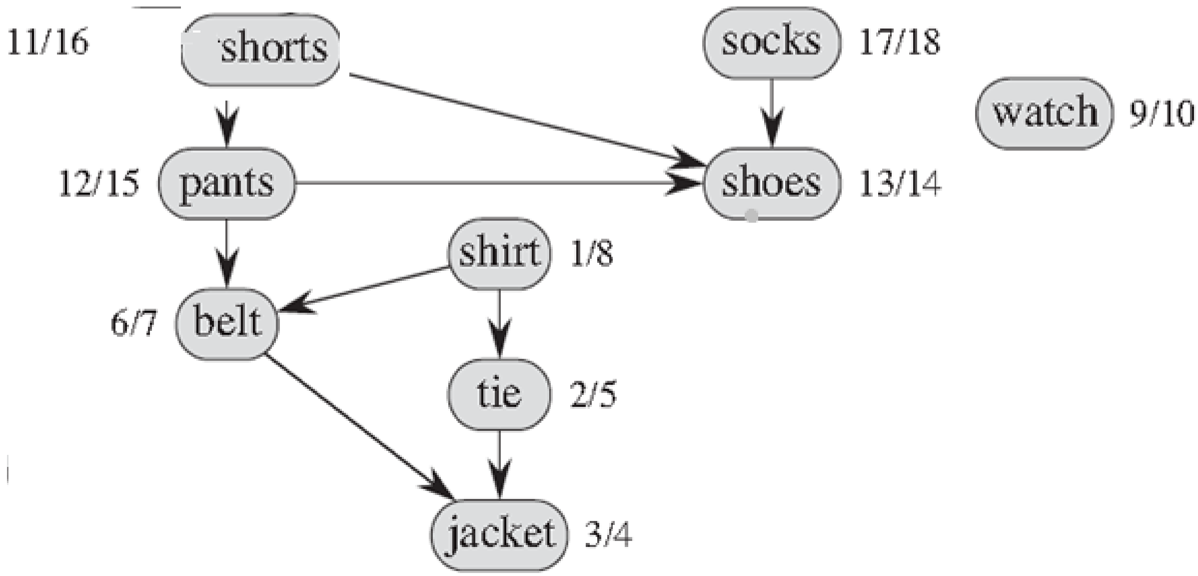

- There can be several orderings of the vertices that fulfill the topological sorting condition:
  - u, v, w, y, x, z
  - w, z, u, v, y, x
  - w, u, v, y, x, z
  - …

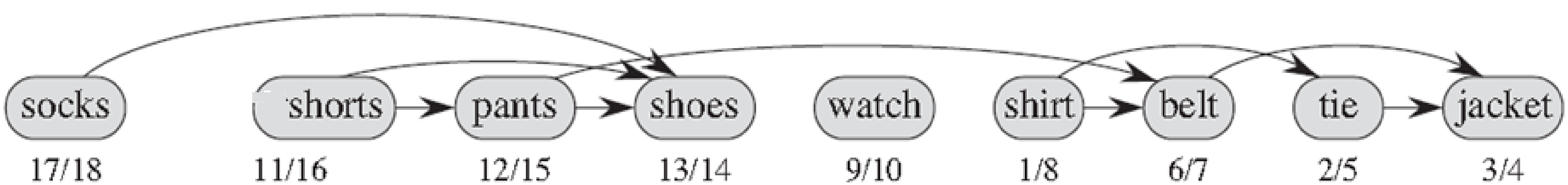
# Topological Sorting

- *Algorithm principle:*
  1. *Call DFS to compute finishing time v.f  for every vertex*
  2. *As every vertex is finished (BLACK) insert it onto the front of a linked list*
  3. *Return the list as the linear ordering  of vertices*

- Time: O(V+E)

# Using DFS for Topological Sorting



# Applications of DFS

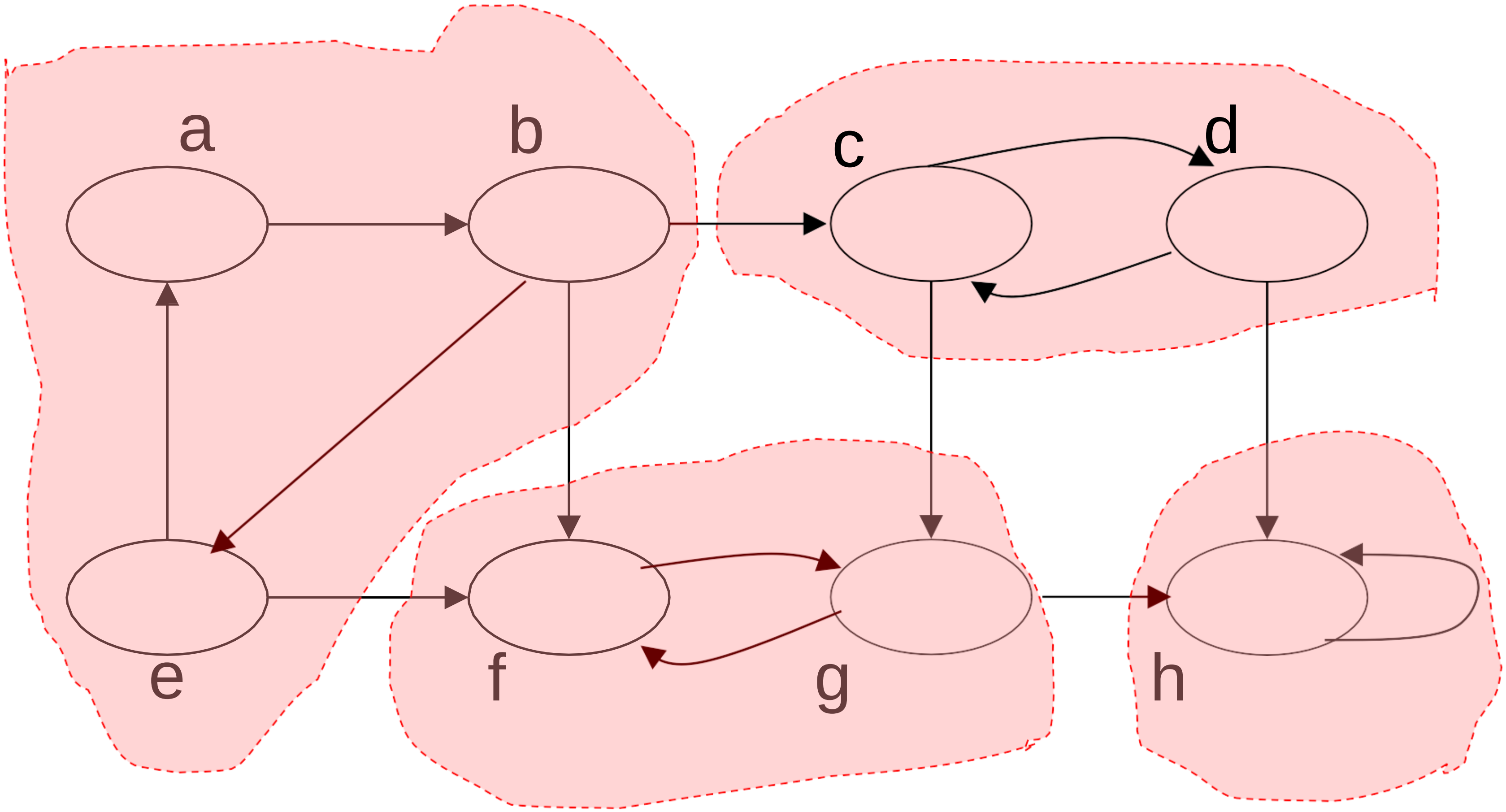
- DFS has many applications
- For undirected graphs:
  - Connected components
  - Connectivity properties
- For directed graphs:
  - Finding cycles
  - Topological sorting
  - Connectivity properties: Strongly connected components

# Strongly Connected Components

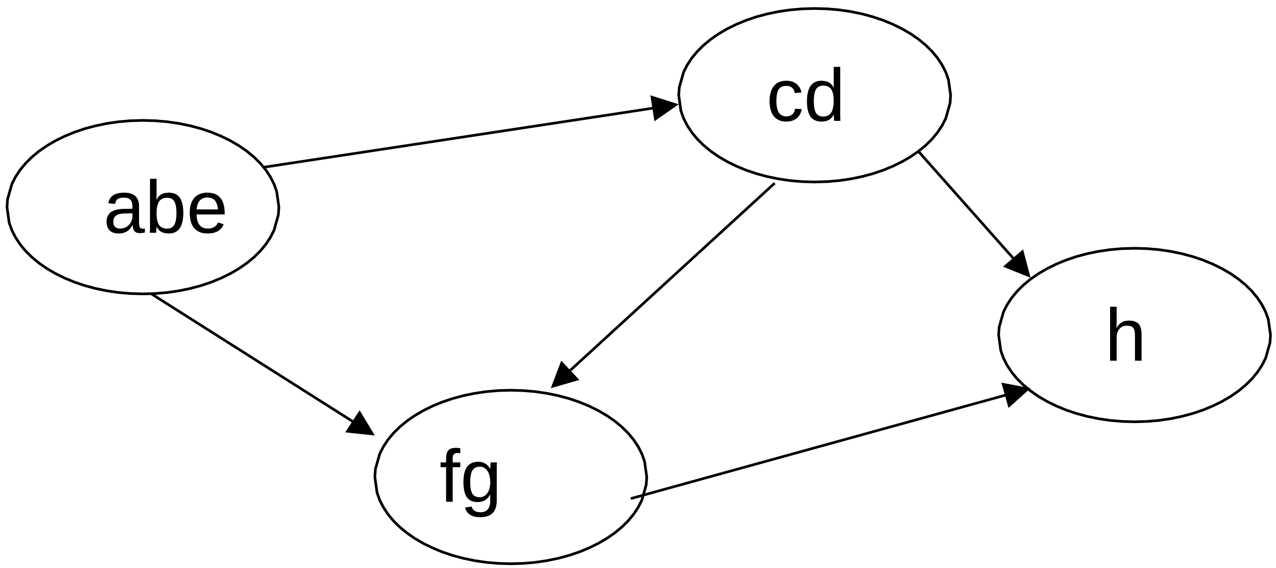
- A strongly connected component of a directed graph G=(V,E) is a maximal set of vertices C such that for every pair of vertices u and v in C, both vertices u and v are reachable from each other.

- KOSARAJU ALGORITHM

# Strongly connected components - Example



# Strongly connected components – Example – The Component Graph




The Component Graph results by collapsing each strong component into a single vertex

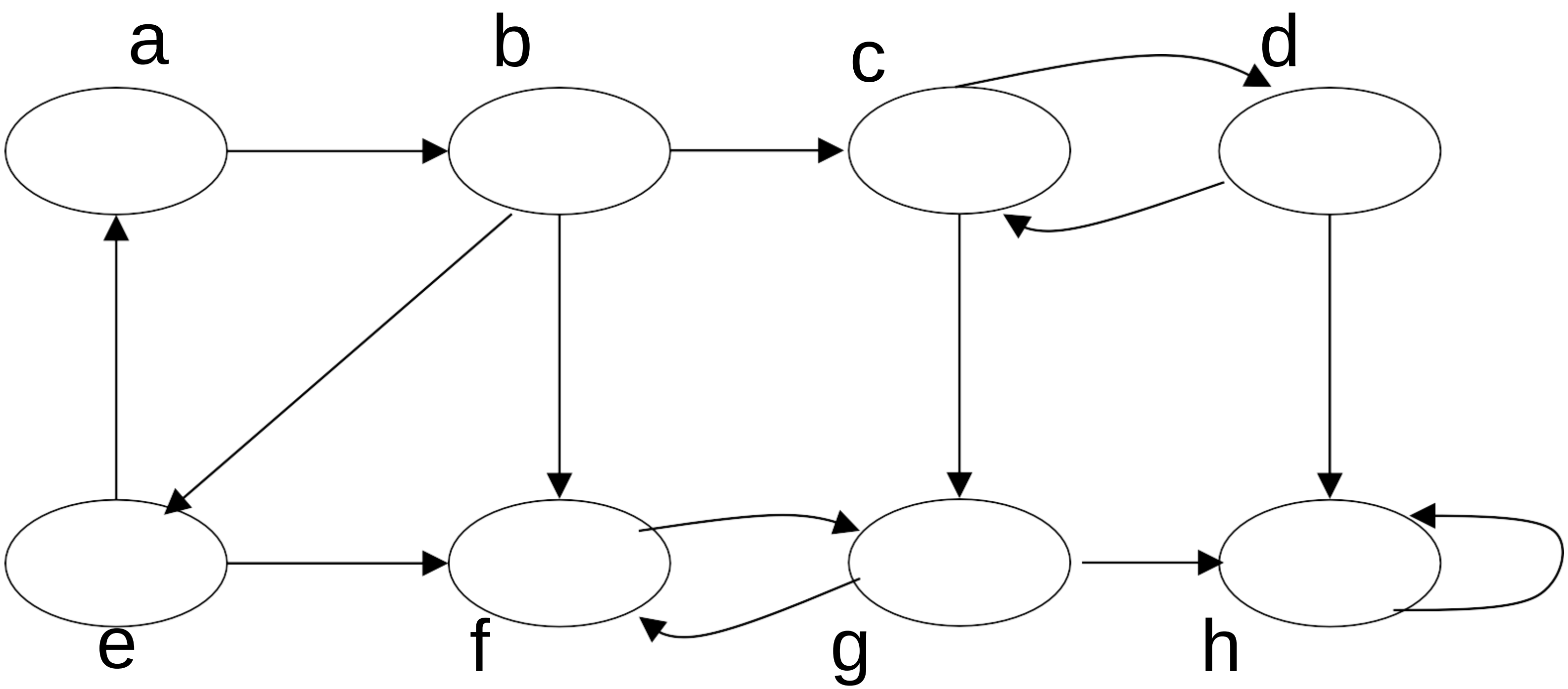
# Strongly connected components

- *Strongly connected components of a directed graph G*
- ***Algorithm principle:***

1. ***Call DFS(G)*** *to compute finishing times u.f for every vertex u*
2. *Compute* ***Graph Transpose (GT)***
3. ***Call DFS(GT),*** *but in the main loop of DFS, consider the vertices in order of decreasing u.f as computed in step 1*
4. *Output the vertices of each DFS-tree formed in step 3 as the vertices of a strongly connected component. (Note: When there is no reachability, we make a manual transition. Each manual transition tell us that a new component is starting.)*

# Strongly connected components - Example

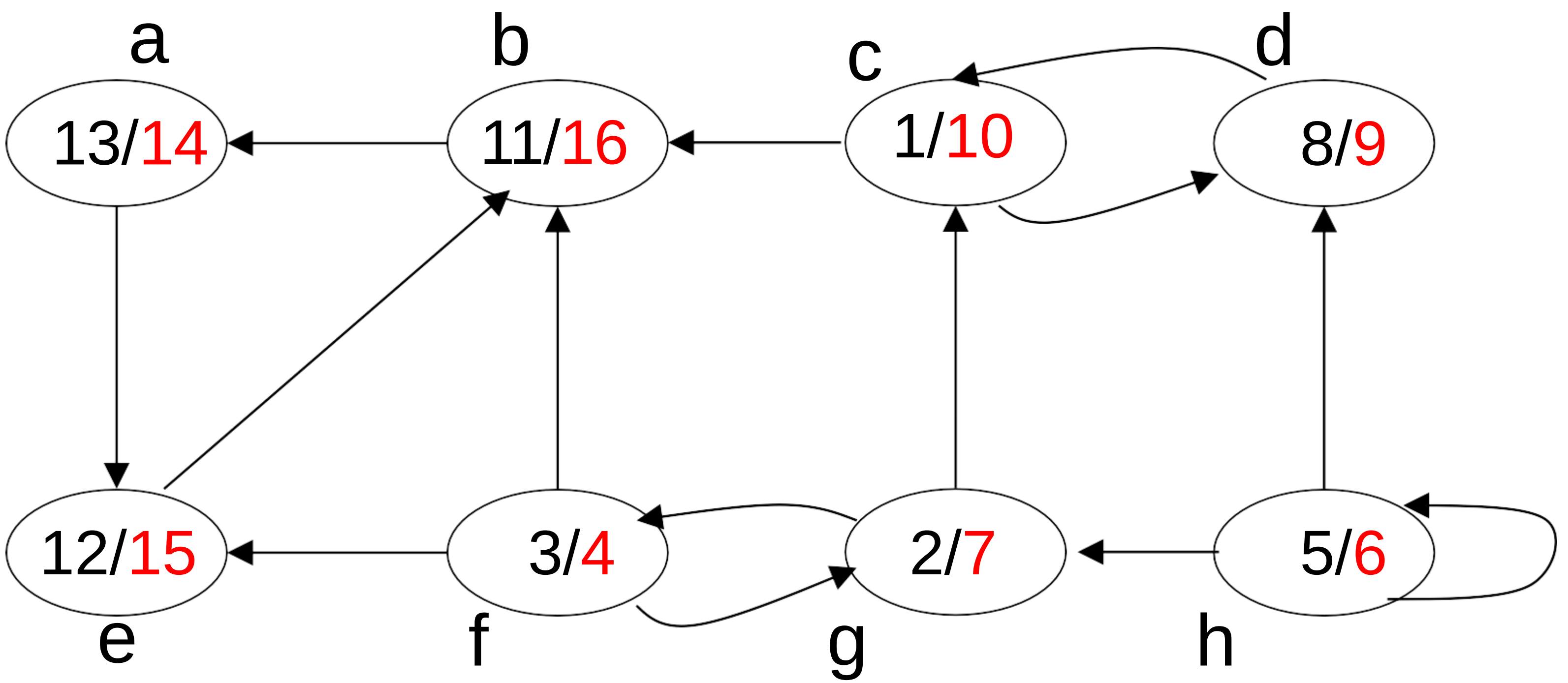
Step1: call DFS(G), compute u.f  for all u. Say I start with ‘c’




| Node | a | b | c | d | e | f | g | h |
|---|---|---|---|---|---|---|---|---|
| Vis | F | F | F | F | F | F | F | F |

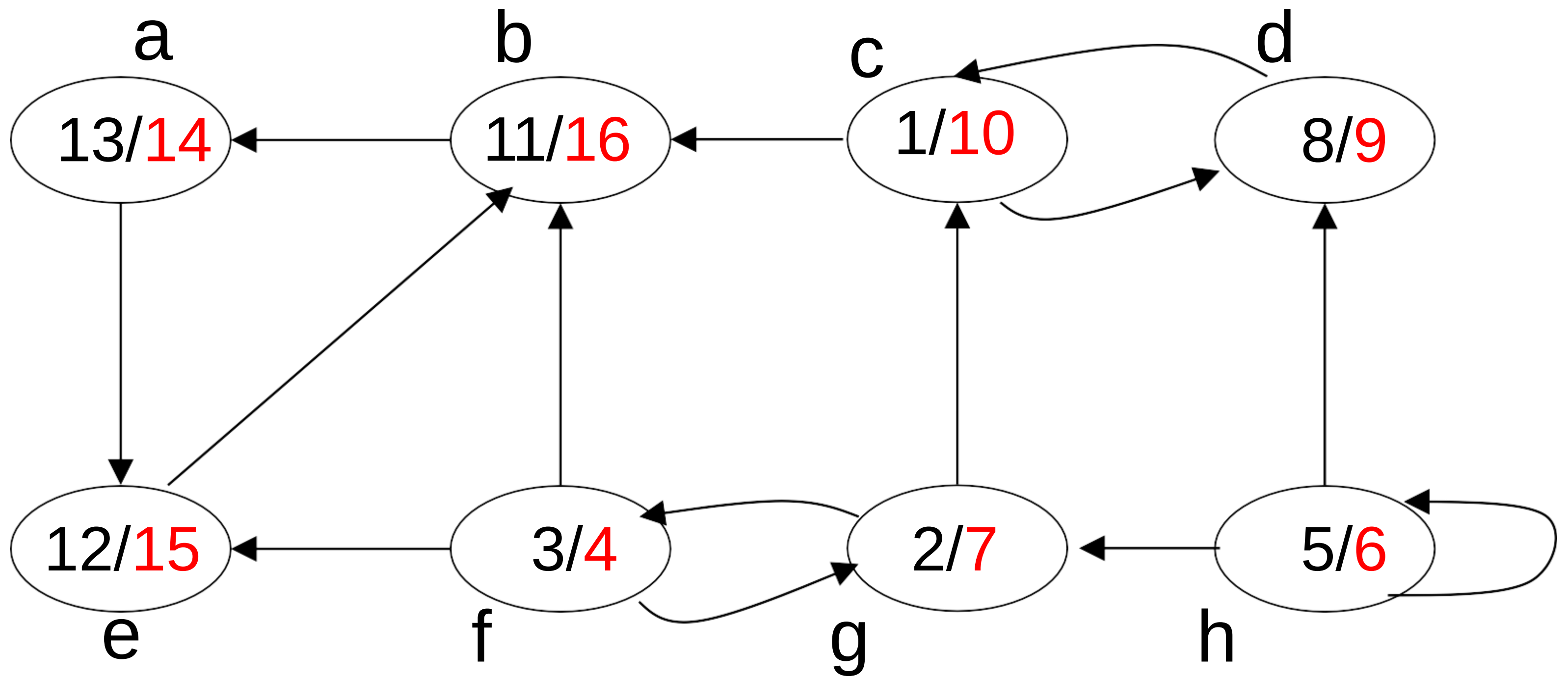
# Strongly connected components - Example

Step2: compute GT



# Strongly connected components - Example

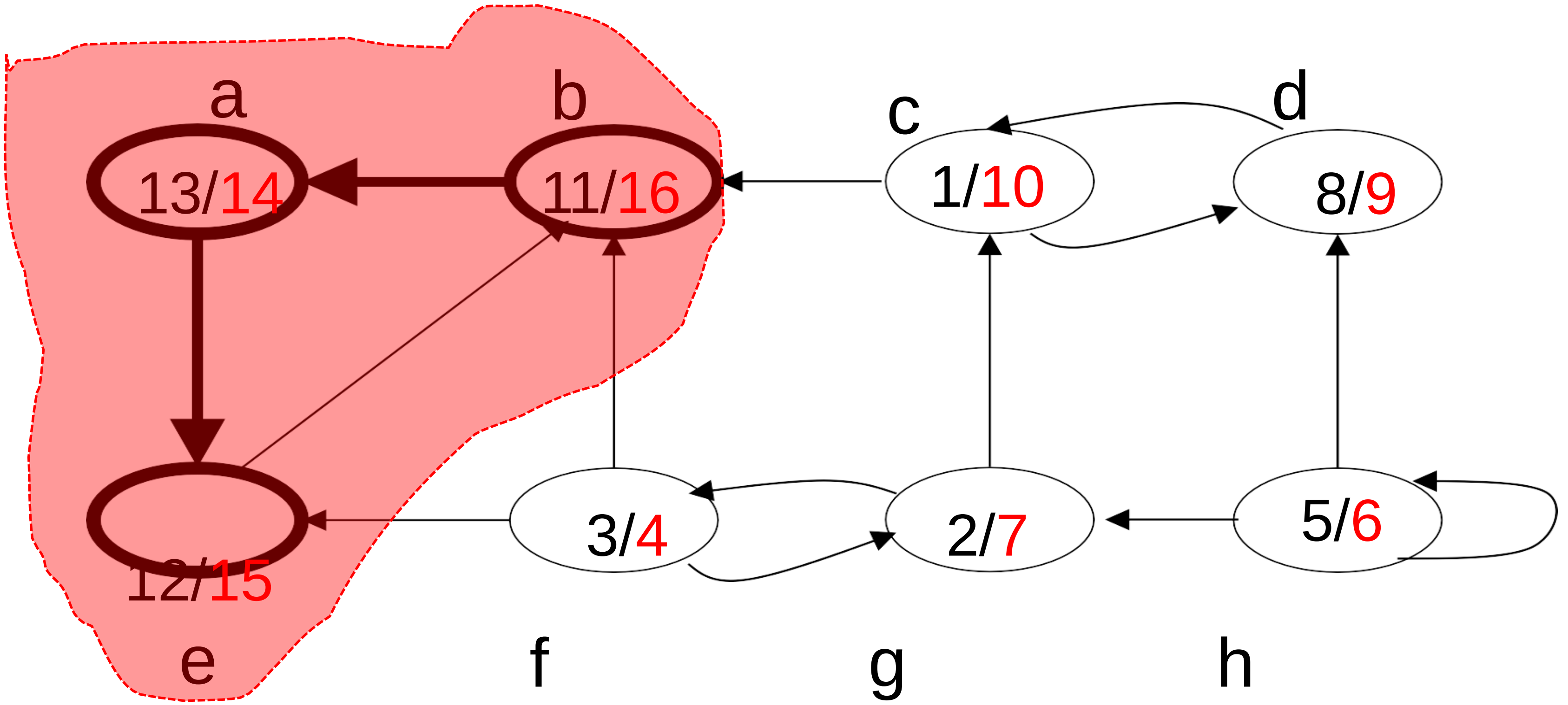
Step3: call DFS(GT), consider vertices in order of decreasing u.f




| Node | a | b | c | d | e | f | g | h |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| u.f | 14 | 16 | 10 | 9 | 15 | 4 | 7 | 6 |

# Strongly connected components - Example

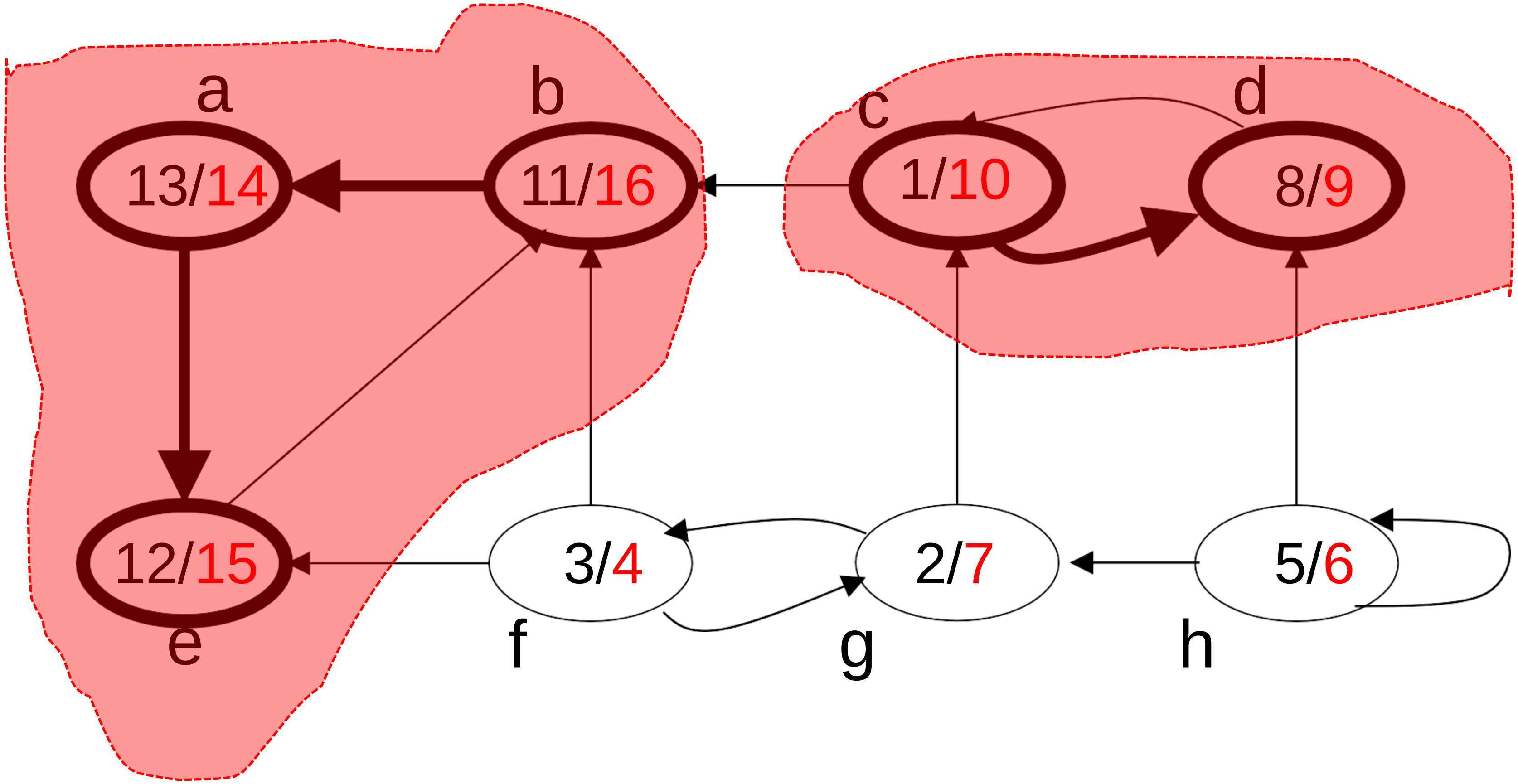
Step3: call DFS(GT), consider vertices in order of decreasing u.f




| Node | a | b | c | d | e | f | g | h |
|---|---|---|---|---|---|---|---|---|
| u.f | 14 | 16 | 10 | 9 | 15 | 4 | 7 | 6 |

# Strongly connected components - Example

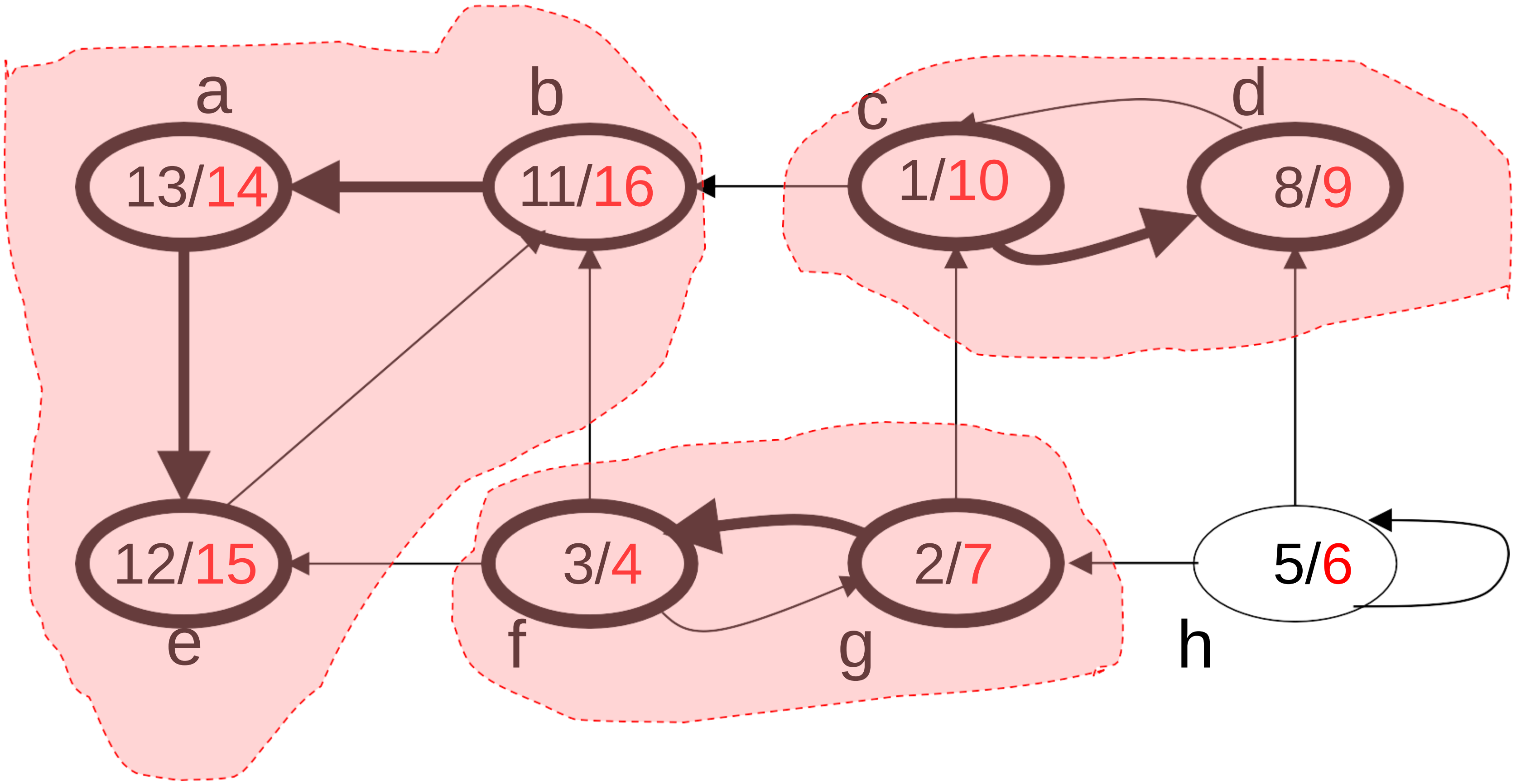
Step3: call DFS(GT), consider vertices in order of decreasing u.f




| Node | a | b | c | d | e | f | g | h |
|---|---|---|---|---|---|---|---|---|
| u.f | 14 | 16 | 10 | 9 | 15 | 4 | 7 | 6 |

# Strongly connected components - Example

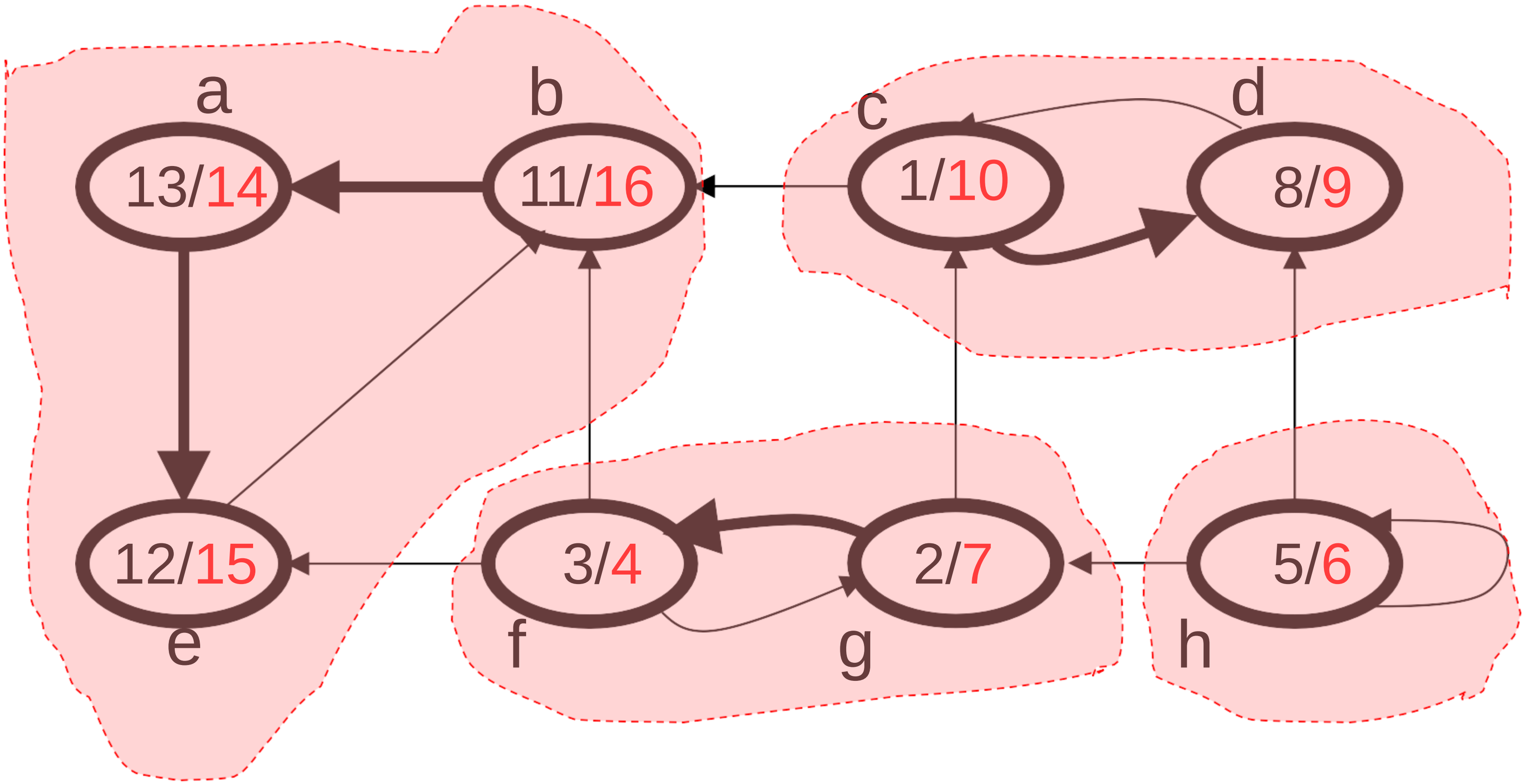
Step3: call DFS(GT), consider vertices in order of decreasing u.f




| Node | a | b | c | d | e | f | g | h |
|---|---|---|---|---|---|---|---|---|
| u.f | 14 | 16 | 10 | 9 | 15 | 4 | 7 | 6 |

# Strongly connected components - Example

Step3: call DFS(GT), consider vertices in order of decreasing u.f




| Node | a | b | c | d | e | f | g | h |
|---|---|---|---|---|---|---|---|---|
| u.f | 14 | 16 | 10 | 9 | 15 | 4 | 7 | 6 |