Articulation Point & Bridges

Articulation Point: A vertex of a graph whose removal increases the number of connected components or converts the directed graph into a disconnected graph.
Bridge: An edge of a graph whose removal increases the number of connected components or converts the directed graph into a disconnected graph.

Approach to identify Bridges and Articulation point using DFS:
Delete a vertex or an edge of a graph and then apply DFS and keep the count of connected components. If the number of components in the graph are greater than 1 then it means that the deleted vertex/edge is an articulation point/bridge respectively.

```
Articulation Point(G)
{
  for each vertex "v" belongs to G.V
    v. color = white
  for each vertex "v" belongs to G.V
  {
    //save the original graph before deletion.
    G' (V, E) = G (V, E)
    delete(v) //delete all the vertices one by one
    compNo = 0
    // Apply DFS after removing a vertex.
    for each vertex "u" belongs G.V
    {
      if (u. color == white)
      {
        DFS-Visit (G, u, ++compNo)
      }
    }
    if (comp No > 1)
      vertex "v" is an articulation point.
 //restore the original graph for next iteration
    G (V, E) = G'(V, E)
  }
}
Time complexity: V*(V+E)
Since E = V^2
```

$$O(V\char94 3)$$

```
Bridge(G)
{
  for each vertex "v" belongs to G.V
    v. color = white
// delete edges one by one instead of vertices
  for each edge "u, v" belongs to G.E
  {
    delete (u, v)
    G'(V, E) = G (V, E)
    compNo = 0
    for each vertex "y" belongs G.V
    {
      if (y. color == white)
      {
        DFS-Visit (G, y, ++compNo)
      }
    }
    if (compNo > 1)
      Edge "u, v" is a bridge.

    G (V, E) = G' (V, E)
  }
}
Time complexity: E*(V+E)
Since E = V^2
```

$$O(V\char94 4)$$

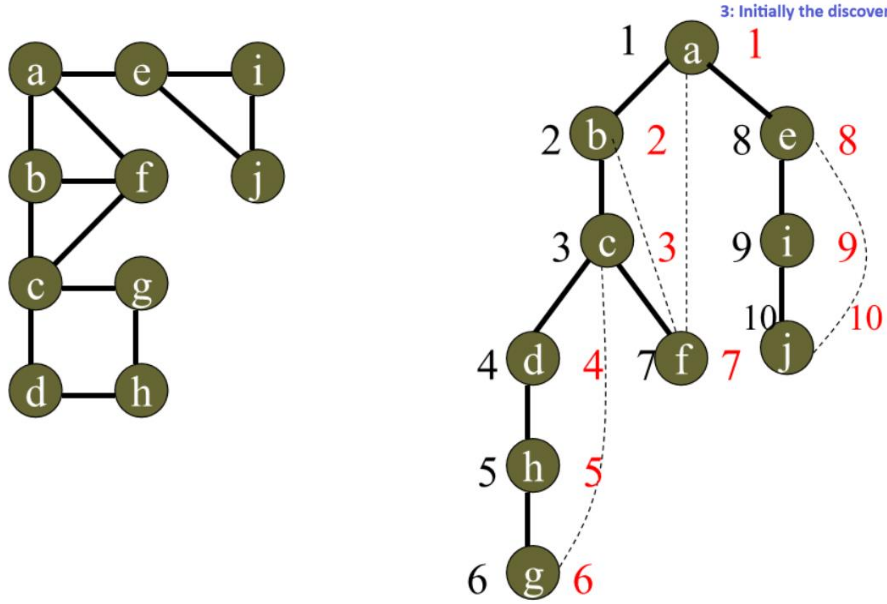
```
DFS-Visit (G, src, compNo){
  src. componentNumber = compNo
  src. color = gray
  for each vertex "v" belongs to G. Adj[src]
  {
    if (v. color == white)
      DFS-Visit (G, v, compNo)
  }
}
```

Tarjan's method is an efficient approach to identify bridges and articulation points in graph. It uses an attribute low value to determine whether a vertex or edge is an articulation point or bridge. Low value will be updated in two scenarios **(back edge & back track)**

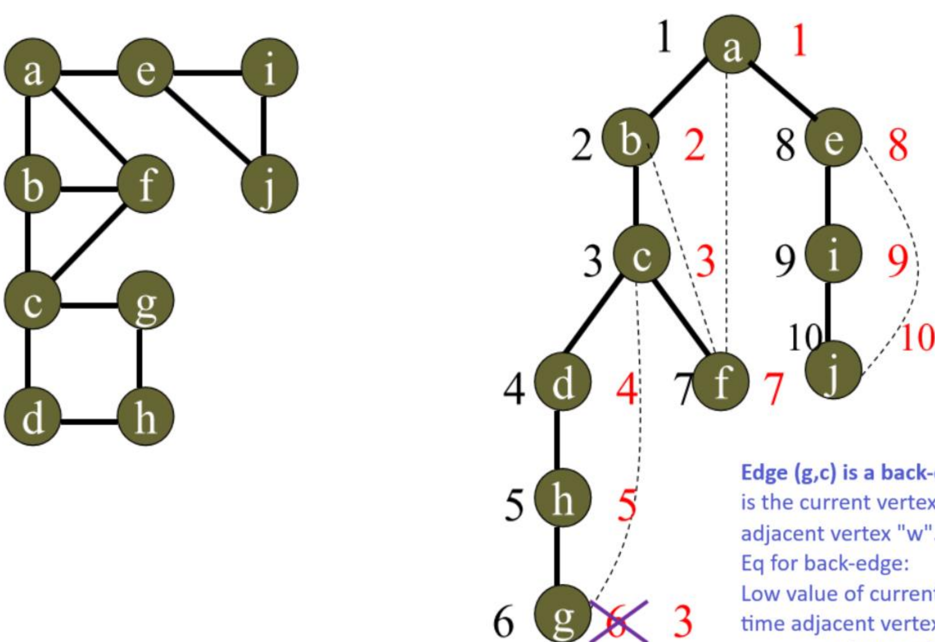
```
Tarjan's Algo(G)
{
   for each vertex "v" belongs to G.V
      v. color = white
   for each vertex "u" belongs G.V
   {
      if (u. color == white)
      {
         DFS-Visit (G, u)
      }
   }
}
time = 0
DFS-Visit (G, v)
{
   time++
   v. d = time
   v. low = v. d   // initially the low value and discovery time will be the same
   v. color = gray
   for each "w" belongs to G. Adj[v]
   {
      if (w. color == white) // unvisited vertex
      {
         w. PI = v
         DFS-visit (G, w)
         //backtrack case i.e., control returned from a recursive call
         //Check whether "v" is an articulation point or not. How?
         if (v. d <= w. low) //compare discovery time of current vertex with low value of adjacent vertex
            "v" is an articulation point.
         //Check whether the edge (v, w) is a bridge or not
         if (v. d < w. low) //compare discovery time of current vertex with low value of adjacent vertex
            "v, w" is a bridge.
         //update the low value of current vertex.
         v. low = min (v. low, w. low)
      }

      else if (w. color == gray AND w. PI != v) //back-edge
      {
         //update the low value
         v. low = min (v. low, u. d)
      }
   }
}
Time Complexity: O(V + E)
```

Example

1: DFS is applied using a vertex "a" as source vertex and alphabetical order is followed when there are multiple adjacent vertices of any vertex.
2: Values written in black are representing the discovery time and red representing the low value.
3: Initially the discovery time and low values are equal.



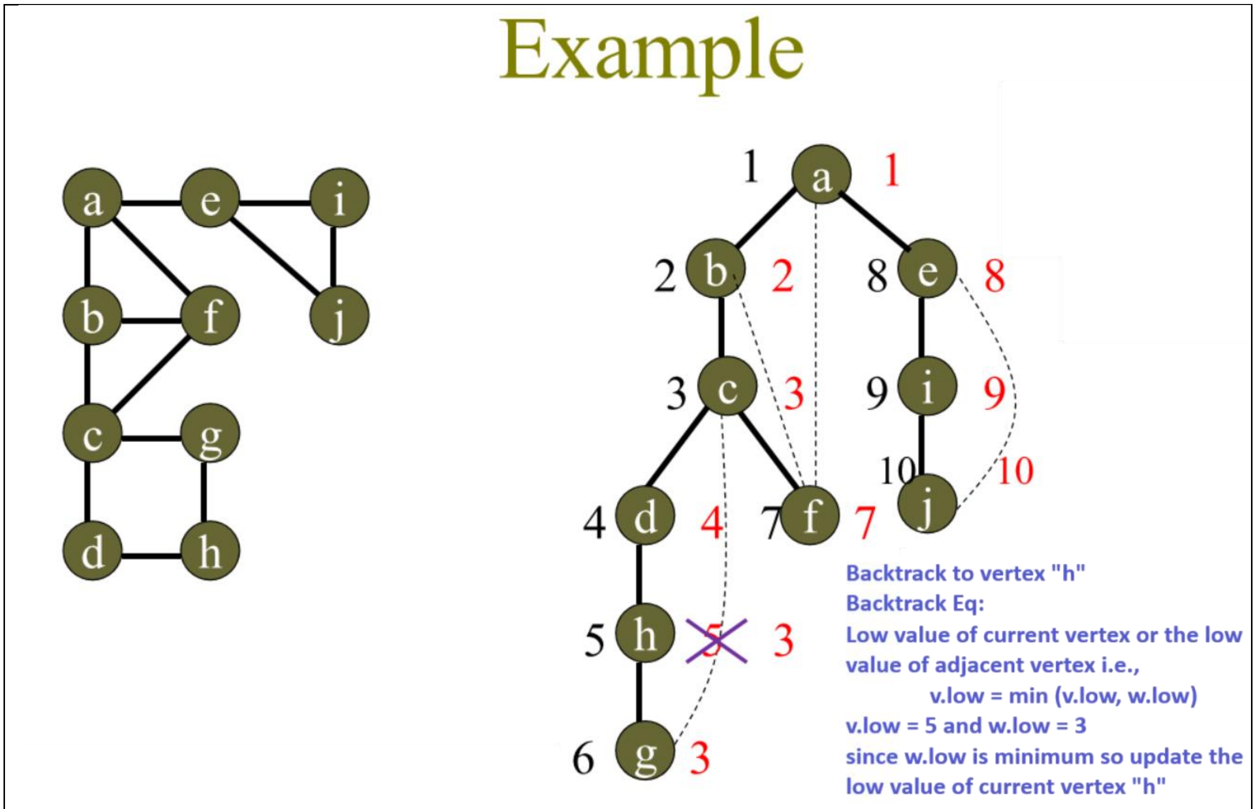
Example

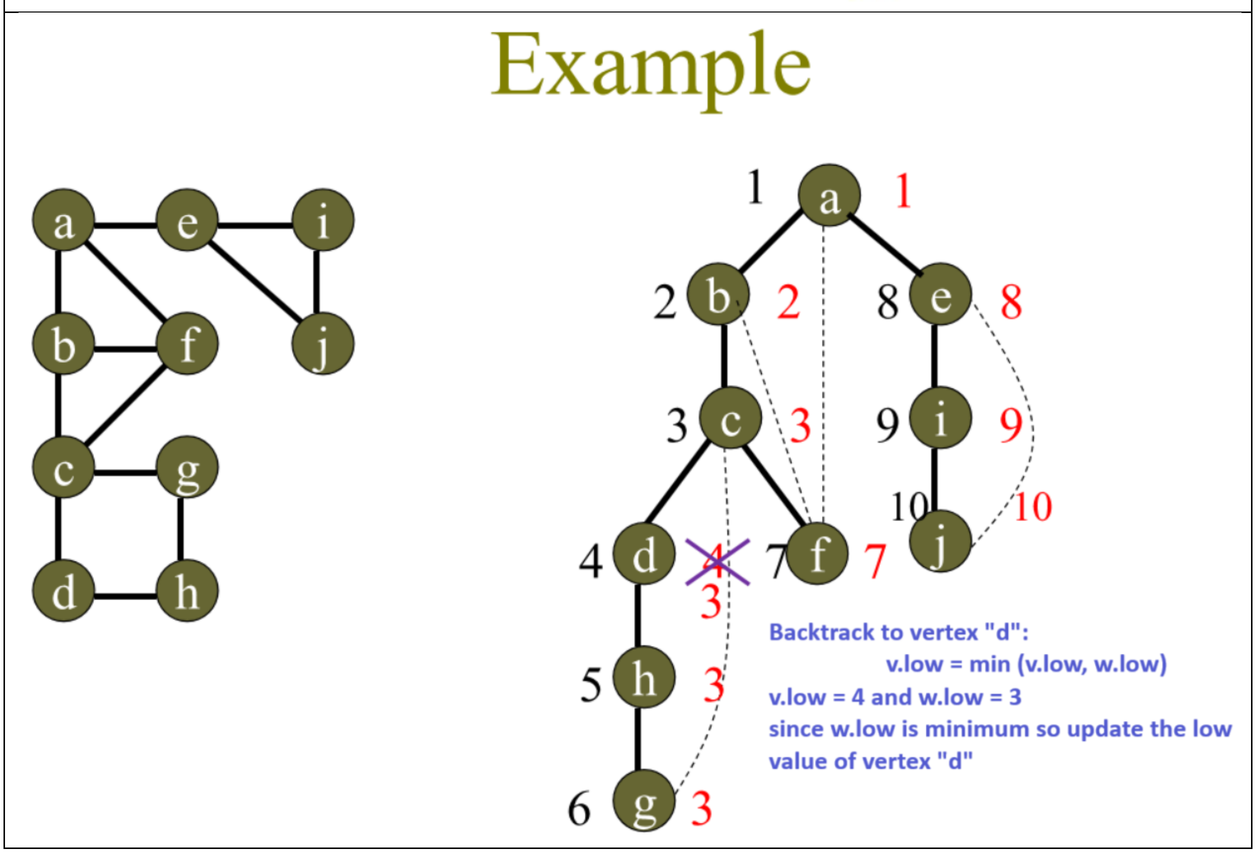


Edge (g,c) is a back-edge where vertex "g" is the current vertex "v" and "c" is an adjacent vertex "w".
Eq for back-edge:
Low value of current vertex or discovery time adjacent vertex i.e.,

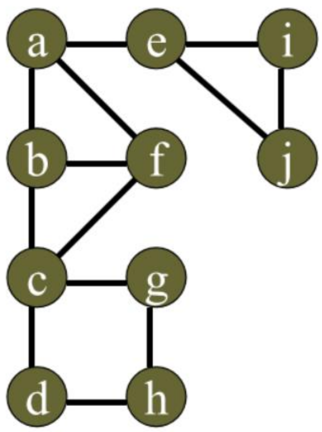
v.low = min(v.low, w.d)

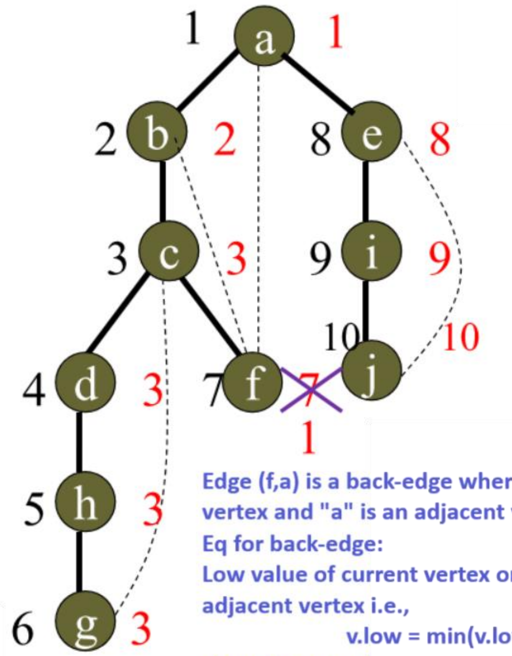
v.low = 6 and w.d = 3
since w.d is minimum so update v.low


Example
Backtrack to vertex "h"
Backtrack Eq:
Low value of current vertex or the low value of adjacent vertex i.e.,
v.low = min (v.low, w.low)
v.low = 5 and w.low = 3
since w.low is minimum so update the low value of current vertex "h"


Example
Backtrack to vertex "d":
v.low = min (v.low, w.low)
v.low = 4 and w.low = 3
since w.low is minimum so update the low value of vertex "d"

Example



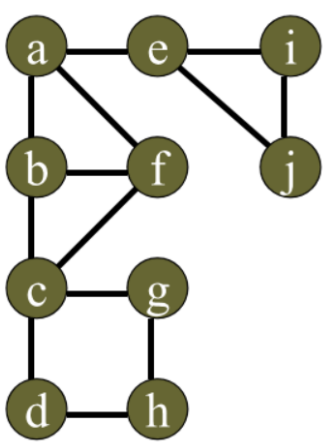


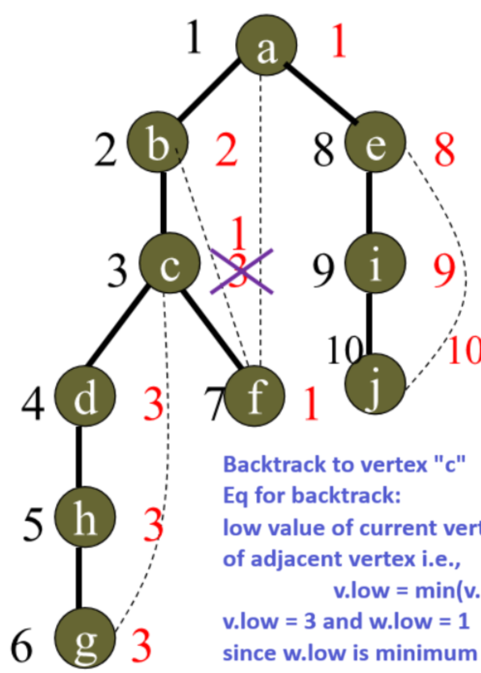
Edge (f,a) is a back-edge where "f" is a current vertex and "a" is an adjacent vertex.
Eq for back-edge:
Low value of current vertex or discovery time of adjacent vertex i.e.,

v.low = min(v.low, w.d)

since discovery time of adjacent is minimum so update the low value of vertex "f"

Example



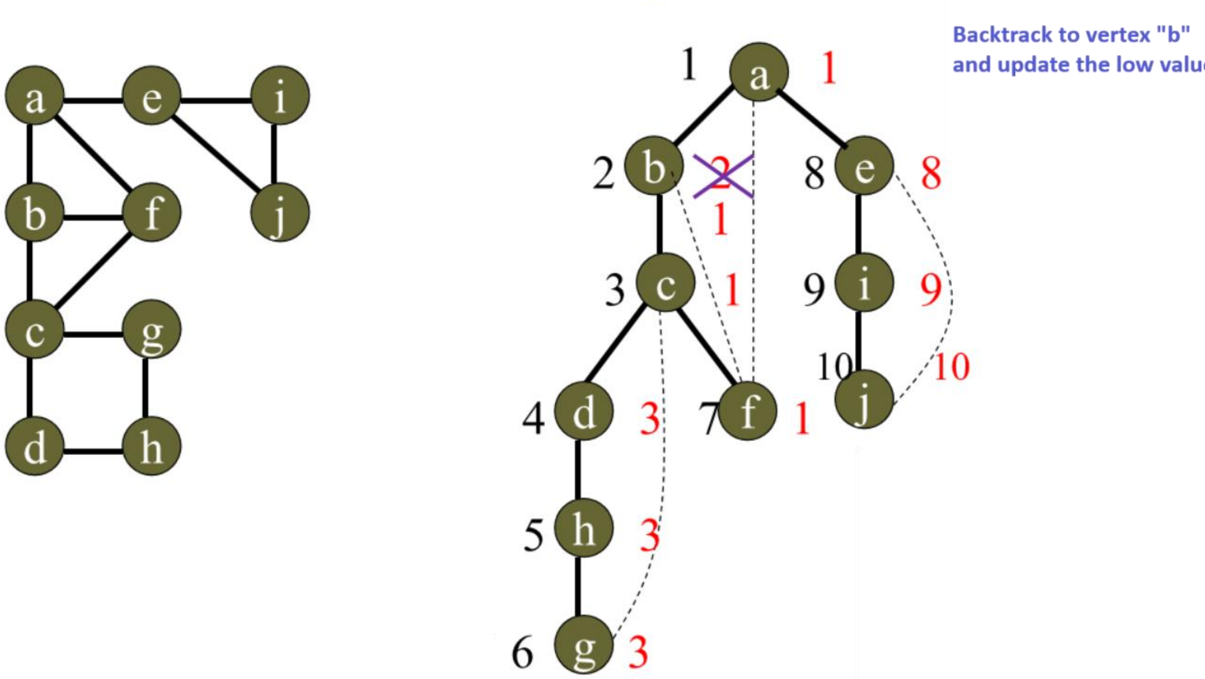


Backtrack to vertex "c"
Eq for backtrack:
low value of current vertex or the low value of adjacent vertex i.e.,

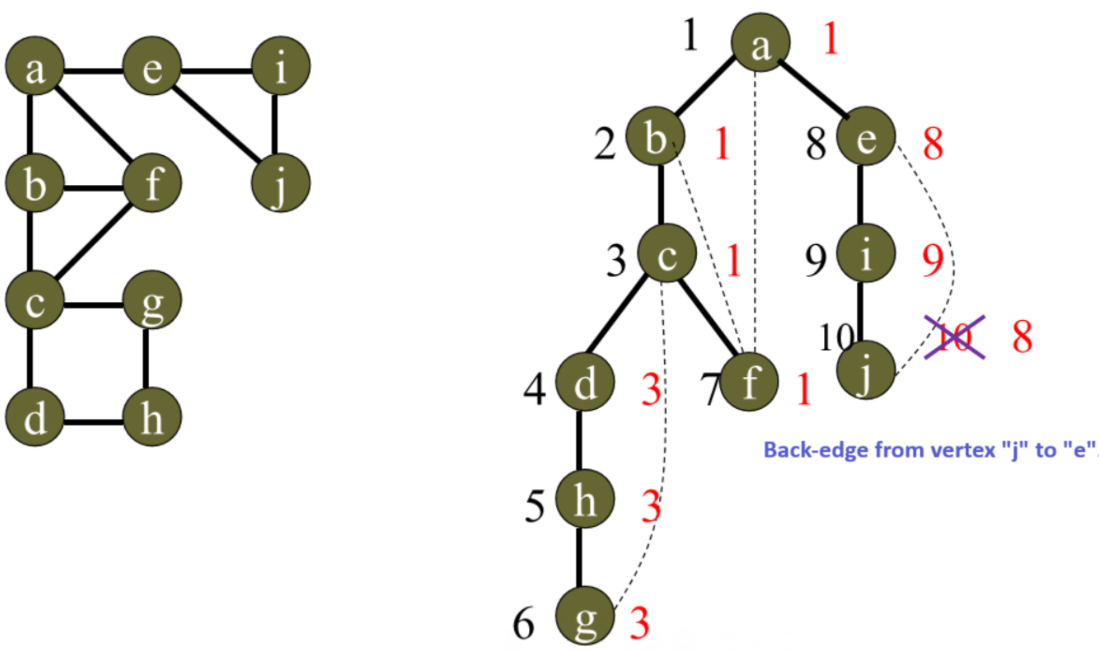
v.low = min(v.low, w.low)

v.low = 3 and w.low = 1
since w.low is minimum so update the value of vertex "c".

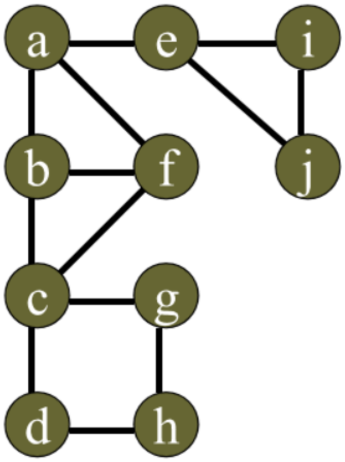
Example

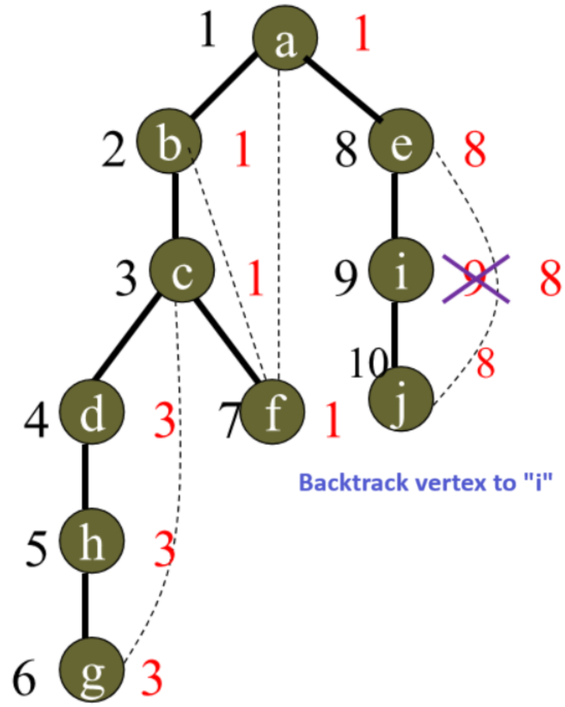


Example

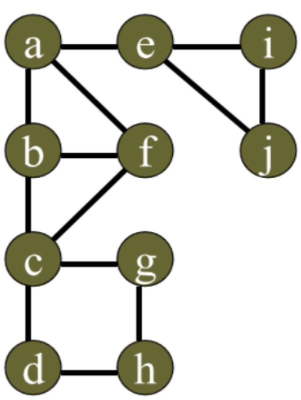


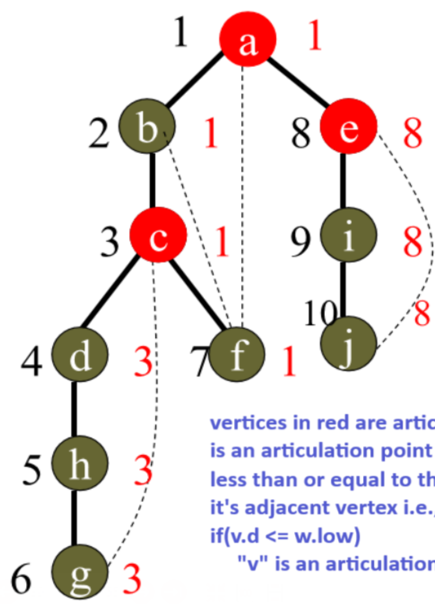
Example





Example





vertices in red are articulation points. A vertex is an articulation point if it's discovery time is less than or equal to the low values of any of it's adjacent vertex i.e.,

if(v.d <= w.low)

"v" is an articulation point