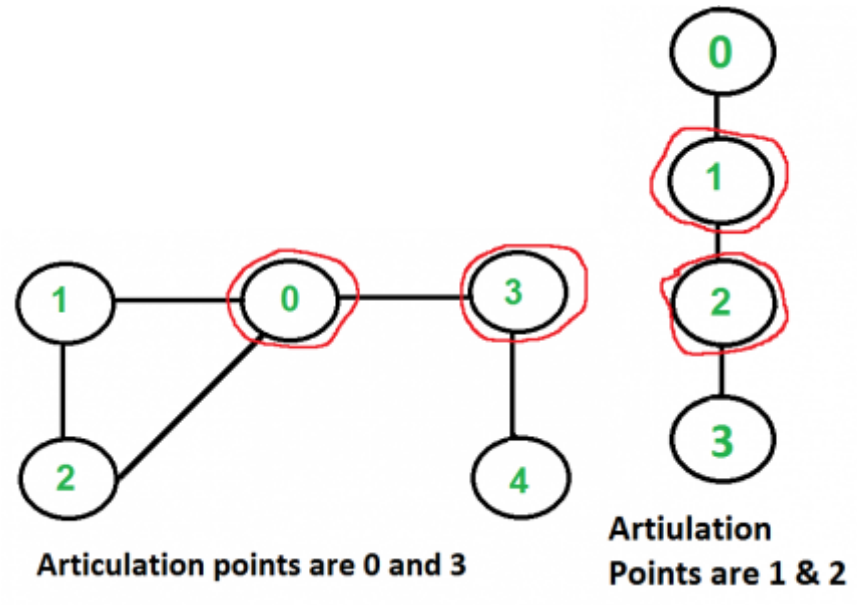
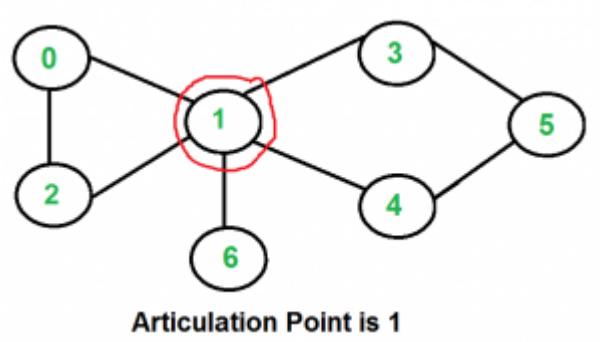
Articulation Points (or Cut Vertices) in a Graph

A vertex 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 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.

Following are some example graphs with articulation points encircled with red color.





How to find all articulation points in a given graph?
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.

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

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

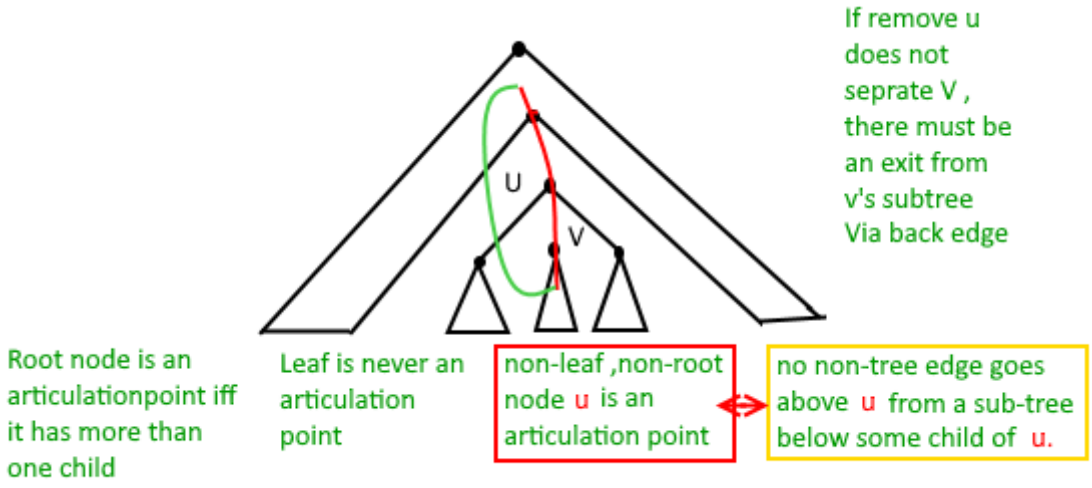
A O(V+E) algorithm to find all Articulation Points (APs)

The idea is to use DFS (Depth First Search). In DFS, we follow vertices in tree form called DFS tree. In DFS tree, a vertex u is parent of another vertex v, if v is discovered by u (obviously v is an adjacent of u in graph). In DFS tree, a vertex u is articulation point if one of the following two conditions is true.

1) u is root of DFS tree and it has at least two children.

2) u is not root of DFS tree and it has a child v such that no vertex in subtree rooted with v has a back edge to one of the ancestors (in DFS tree) of u.

Following figure shows same points as above with one additional point that a leaf in DFS Tree can never be an articulation point.



We do DFS traversal of given graph with additional code to find out Articulation Points (APs). In DFS traversal, we maintain a parent[] array where parent[u] stores parent of vertex u. Among the above mentioned two cases, the first case is simple to detect. For every vertex, count children. If currently visited vertex u is root (parent[u] is NIL) and has more than two children, print it.

How to handle second case? The second case is trickier. We maintain an array disc[] to store discovery time of vertices. For every node u, we need to find out the earliest visited vertex (the vertex with minimum discovery time) that can be reached from subtree rooted with u. So we maintain an additional array low[] which is defined as follows.

```
low[u] = min(disc[u], disc[w])
where w is an ancestor of u and there is a back edge from
some descendant of u to w.
```

Following is the implementation of Tarjan's algorithm for finding articulation points.

Time Complexity: The above function is simple DFS with additional arrays. So time complexity is same as DFS which is O(V+E) for adjacency list representation of graph.

```
from collections import defaultdict


def articulationPoints(edges):
    graph = defaultdict(list)
    for u, v in edges:
        graph[u].append(v)
        graph[v].append(u)
    dfs(0, graph, 0)


def dfs(u, graph, time):
    dis[u] = time
    low[u] = time
    time = time + 1
    visited[u] = True
    count = 0
    for v in graph[u]:
        if parent[u] == v:
            continue
        elif visited[v] is True:
            low[u] = min(low[u], dis[v])
        else:
            parent[v] = u
            count = count+1
            dfs(v, graph, time)
            low[u] = min(low[u], low[v])
            if parent[u]==-1:
                if count>=2:
                    ap[u] = True
            else:
                if low[v]>=dis[u]:
                    ap[u] = True


edges = [(0, 1),(1,2),(2,0),(2,3),(2,4),(4,3),(3,5),(4,6),(1,8),(0,7)]
parent = [-1] * len(edges)
dis = [-1] * len(edges)
low = [-1] * len(edges)
visited = [False] * len(edges)
ap = [False] * len(edges)
time = 0
```

```
articulationPoints(edges)
print(ap)
```