<https://emre.me/algorithms/tarjans-algorithm/>

**Tarjan’s algorithm**

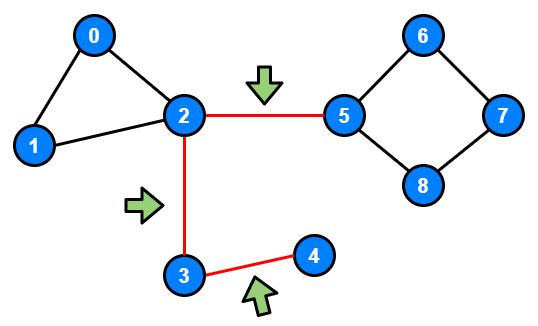
[1](https://emre.me/algorithms/tarjans-algorithm/#references), [2](https://emre.me/algorithms/tarjans-algorithm/#references) which runs in *linear time* is an algorithm in [Graph Theory](https://emre.me/data-structures/graphs/) for finding the **strongly connected components** of a [directed graph](https://emre.me/data-structures/graphs/#directed-or-undirected).

**Bridges and Articulation Points**

**Bridges** and **Articulation Points** are important in [Graph Theory](https://emre.me/data-structures/graphs/) because in real-world situations, they often hint *weak points*, *bottle necks* or *vulnerabilities* in the graph. Therefore, it is important to be able to quickly *find* and *detect* **where** and **when** they occur.

**Bridge**

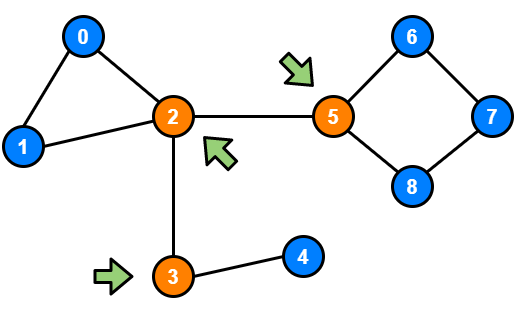
A [Bridge](https://en.wikipedia.org/wiki/Bridge_(graph_theory))(or **cut-edge**) in [graph theory](https://emre.me/data-structures/graphs/) is any **edge** in a graph *whose removal increases the number of connected components*.



Lines with the *red color* are **bridges** because if you remove any of them, the graph is *divided into two components*.

**Articulation Point**

An [Articulation Point](https://en.wikipedia.org/wiki/Biconnected_component)(or **cut-vertex**) is any **node** in a graph *whose removal increases the number of connected components*.



Nodes marked with *orange color* are **articulation points** because if you remove any of them, graph is *divided into two components*.

**Tarjan’s Algorithm**

[Tarjan’s Algorithm](https://en.wikipedia.org/wiki/Tarjan%27s_strongly_connected_components_algorithm) provides a very effective way to find these **bridges** and **articulation points** in linear time. We can explain this algorithm in **3** steps:

**Step 1**

Start at **any** *node* and do a [Depth First Search (DFS)](https://emre.me/coding-patterns/depth-first-search/) traversal, labeling nodes with an *increasing* id value as you go.



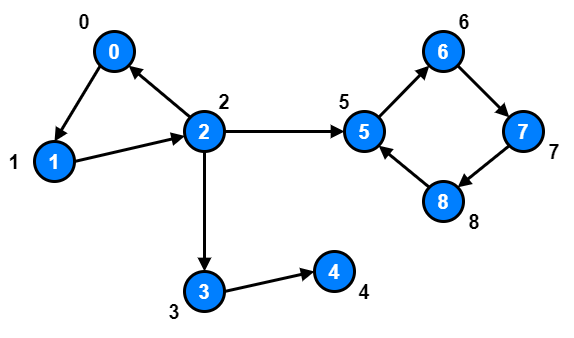
**Step 2**

Keep track the id of *each* node and the *smallest* [low link](https://emre.me/algorithms/tarjans-algorithm/#what-is-low-link) value.

**What is Low Link?**

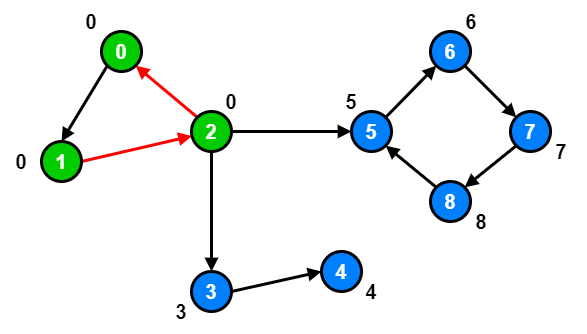
**Low Link Value** of a node is defined as the smallest id reachable from that node when doing a [Depth First Search (DFS)](https://emre.me/coding-patterns/depth-first-search/), including itself.

Initially, all [low link](https://emre.me/algorithms/tarjans-algorithm/#what-is-low-link) values can be initialized to the each node id.

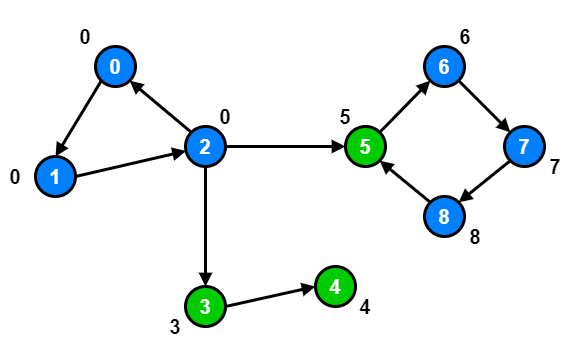


If we inspect *node 1* and *node 2*, we will notice that **there exist a path** going from *node 1* and *node 2* to *node 0*.

So, we should update both *node 1* and *node 2* [low link](https://emre.me/algorithms/tarjans-algorithm/#what-is-low-link) values to **0**.

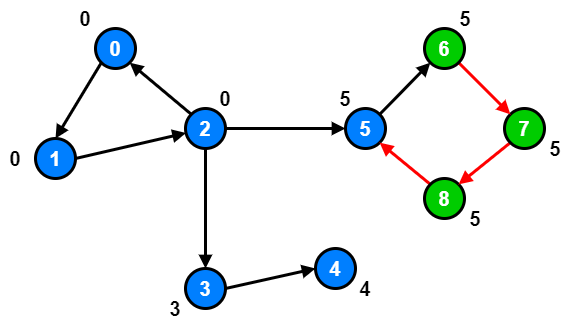


However, *node 3*, *node 4* and *node 5* are already at their optimal [low link](https://emre.me/algorithms/tarjans-algorithm/#what-is-low-link) value because there are **no other node** they can reach with a smaller id.



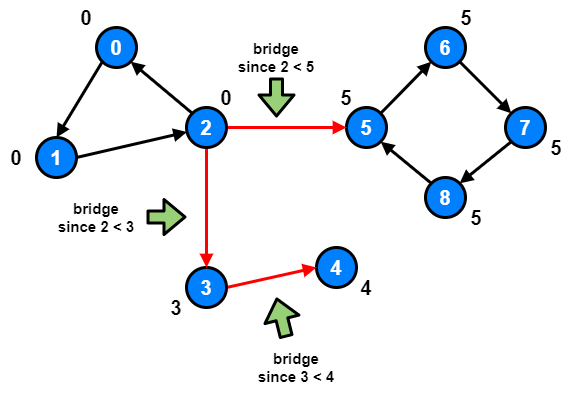
For *node 6*, *node 7* and *node 8*, **there is a path** from *node 6*, *node 7* and *node 8*  to *node 5*.

So, we should update *node 6*, *node 7* and *node 8* [low link](https://emre.me/algorithms/tarjans-algorithm/#what-is-low-link) values to **5**.



**Step 3**

During the [Depth First Search (DFS)](https://emre.me/coding-patterns/depth-first-search/), bridges will be found where the id of node your edge is coming from is **less than** the [low link](https://emre.me/algorithms/tarjans-algorithm/#what-is-low-link) value of the node your edge is going to.



**Problem: Critical Connections in a Network**

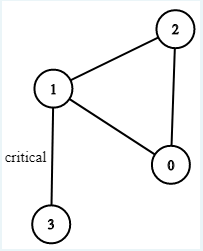
**LeetCode 1192 - Critical Connections in a Network** [[](https://leetcode.com/problems/critical-connections-in-a-network/)*hard*[]](https://leetcode.com/problems/critical-connections-in-a-network/)

There are n servers numbered from **0** to **n-1** connected by undirected server-to-server connections forming a network where connections[i] = [a, b] represents a connection between servers **a** and **b**. Any server can reach any other server directly or indirectly through the network.

A critical connection is a connection that, if removed, will make some server unable to reach some other server.

Return all critical connections in the network in any order.

**Example 1:**



Input: n = 4, connections = [[0, 1], [1, 2], [2, 0], [1, 3]]

Output: [[1, 3]]

Explanation: [[3, 1]] is also accepted.

**Constraints:**

1. 1 <= n <= 10^5
2. n-1 <= connections.length <= 10^5
3. connections[i][0] != connections[i][1]
4. There are no repeated connections.

**Solution with using Tarjan’s Algorithm**

from collections import defaultdict

from typing import List

class Solution:

    def criticalConnections(self, n: int, connections: List[List[int]]) -> List[List[int]]:

        def make\_graph(connections):

            graph = defaultdict(list)

            for edge in connections:

                a, b = edge

                graph[a].append(b)

                graph[b].append(a)

            return graph

        graph = make\_graph(connections)

        id, node, prev\_node = 0, 0, -1  # at first there is no prev\_node. we set it to -1

        ids = [0 for \_ in range(n)]  # tracks ids of nodes

        low\_links = [0 for \_ in range(n)]  # tracks low link value (default value is the index)

        visited = [False for \_ in range(n)]  # tracks DFS visit status

        bridges = []

        self.dfs(node, prev\_node, bridges, graph, id, visited, ids, low\_links)

        return bridges

    def dfs(self, node, prev\_node, bridges, graph, id, visited, ids, low\_links):

        visited[node] = True

        low\_links[node] = id

        ids[node] = id

        id += 1

        for next\_node in graph[node]:

            if next\_node == prev\_node:

                continue

            if not visited[next\_node]:

                self.dfs(next\_node, node, bridges, graph, id, visited, ids, low\_links)

                low\_links[node] = min(low\_links[node], low\_links[next\_node])  # on callback, generates low link values

                if ids[node] < low\_links[next\_node]:  # found the bridge!

                    bridges.append([node, next\_node])

            else:

                # tried to visit an already visited node, which may have a lower id than the current low link value

                low\_links[node] = min(low\_links[node], ids[next\_node])

**Time Complexity**: **O(E + V)** –> One pass, linear time solution

**Space Complexity**: **O(N)**

**Refer to**

[L1192.Critical Connections in a Network](note://62AF86160F014A43951659054B3FC601)