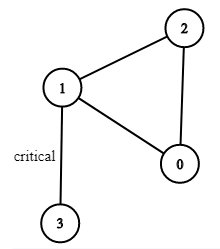
<https://leetcode.com/problems/critical-connections-in-a-network/>

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

A critical connection is a connection that, if removed, will make some servers 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.

**Example 2:**

**Input**: n = 2, connections = [[0,1]]

**Output**: [[0,1]]

**Constraints:**

2 <= n <= 10^5

n - 1 <= connections.length <= 10^5

0 <= ai, bi <= n - 1

ai != bi

There are no repeated connections.

**Attempt 1: 2022-11-16**

**Solution 1: Recursive traversal as DFS (360min)**

class Solution {

    public List<List<Integer>> criticalConnections(int n, List<List<Integer>> connections) {

        List<List<Integer>> result = new ArrayList<List<Integer>>();

        // Build undirected graph

        Map<Integer, List<Integer>> graph = new HashMap<Integer, List<Integer>>();

        for(int i = 0; i < n; i++) {

            graph.put(i, new ArrayList<Integer>());

        }

        for(List<Integer> connection : connections) {

            int from = connection.get(0);

            int to = connection.get(1);

            graph.get(from).add(to);

            graph.get(to).add(from);

        }

        // Store all connections into set

        Set<List<Integer>> connection\_set = new HashSet<List<Integer>>(connections);

        int[] rank = new int[n];

        // Initially fill all node's rank as -2 for dfs to identify visited nodes(node' rank > 0)

        Arrays.fill(rank, -2);

        helper(0, 0, graph, connection\_set, rank);

        return new ArrayList<List<Integer>>(connection\_set);

    }

    /\*\*

    How can "rank" help us with removing cycles?

    Imagine you have a current path of length k during a DFS. The nodes on the path has increasing

    ranks from 0 to kand incrementing by 1. Surprisingly, your next visit finds a node that has a

    rank of p where 0 <= p < k. Why does it happen? Aha! You found a node that is on the current

    search path! That means, congratulations, you found a cycle!

    But only the current level of search knows it finds a cycle. How does the upper level of search

    knows, if you backtrack? Let's make use of the return value of DFS: dfs function returns the

    minimum rank it finds. During a step of search from node curr to its neighbor next, if

    dfs(next) returns something smaller than or equal to rank(curr), then curr knows its neighbor next

    helped it to find a cycle back to curr or curr's ancestor. So curr knows it should discard the

    edge (curr, next) which is in a cycle.

    \*/

    private int helper(int curr, int depth, Map<Integer, List<Integer>> graph, Set<List<Integer>> connection\_set, int[] rank) {

        // Already visited node. return its rank

        if(rank[curr] >= 0) {

            return rank[curr];

        }

        // If not visited before, set current curr's rank is depth,

        // e.g node 0's rank is depth = 0 initially

        rank[curr] = depth;

        // Use minDepthFound to record minimum depth found so far on the dfs path,

        // instead of Integer.MAX\_VALUE, value can be 'depth' also

        int minDepthFound = Integer.MAX\_VALUE;

        // Find all neighbor nodes of current node and start dfs

        for(int next : graph.get(curr)) {

            // If neighbor is parent of current node(rank is depth - 1) ignore

            if(rank[next] == depth - 1) {

                continue;

            }

// If dfs(next) returns something smaller than or equal to rank(curr),

// then curr knows its neighbor next helped it to find a cycle back to

// curr or curr's ancestor. So curr knows it should discard the

// edge(curr, next) which is in a cycle

            int minDepth = helper(next, depth + 1, graph, connection\_set, rank);

            minDepthFound = Math.min(minDepth, minDepthFound);

            if(minDepth <= depth) {

                // To avoid the sorting just try to remove both combinations. of (x,y) and (y,x)

                connection\_set.remove(Arrays.asList(curr, next));

                connection\_set.remove(Arrays.asList(next, curr));

            }

        }

        return minDepthFound;

    }

}

**Refer to**

<https://leetcode.com/problems/critical-connections-in-a-network/discuss/382638/DFS-detailed-explanation-O(orEor)-solution>

**First thought**

Thiking for a little while, you will easily find out this theorem on a connected graph:

**An edge is a critical connection, if and only if it is not in a cycle.**

So, if we know how to find cycles, and discard all edges in the cycles, then the remaining connections are a complete collection of critical connections.

**How to find edges in cycles, and remove them**

We will use DFS algorithm to find cycles and decide whether or not an edge is in a cycle.

Define **rank** of a node: The depth of a node during a DFS. The starting node has a rank 0.

Only the nodes on the current DFS path have non-special ranks. In other words, only the nodes that we've started visiting, but haven't finished visiting, have ranks. So 0 <= rank < n.

(For coding purpose, if a node is not visited yet, it has a special rank -2; if we've fully completed the visit of a node, it has a special rank n.)

**How can "rank" help us with removing cycles?** Imagine you have a current path of length k during a DFS. The nodes on the path has increasing ranks from 0 to kand incrementing by 1. Surprisingly, your next visit finds a node that has a rank of p where 0 <= p < k. Why does it happen? Aha! You found a node that is on the current search path! That means, congratulations, you found a cycle!

But only the current level of search knows it finds a cycle. How does the upper level of search knows, if you backtrack? Let's make use of the return value of DFS: **dfs function returns the minimum rank it finds.** During a step of search from node u to its neighbor v, **if dfs(v) returns something smaller than or equal to rank(u)**, then u knows its neighbor v helped it to find a cycle back to u or u's ancestor. So u knows it should discard the edge (u, v) which is in a cycle.

After doing dfs on all nodes, all edges in cycles are discarded. So the remaining edges are critical connections.

**Python code**

import collections

class Solution(object):

    def criticalConnections(self, n, connections):

        def makeGraph(connections):

            graph = collections.defaultdict(list)

            for conn in connections:

                graph[conn[0]].append(conn[1])

                graph[conn[1]].append(conn[0])

            return graph

        graph = makeGraph(connections)

        connections = set(map(tuple, (map(sorted, connections))))

        rank = [-2] \* n

        def dfs(node, depth):

            if rank[node] >= 0:

                # visiting (0<=rank<n), or visited (rank=n)

                return rank[node]

            rank[node] = depth

            min\_back\_depth = n

            for neighbor in graph[node]:

                if rank[neighbor] == depth - 1:

                    continue  # don't immmediately go back to parent. that's why i didn't choose -1 as the special value, in case depth==0.

                back\_depth = dfs(neighbor, depth + 1)

                if back\_depth <= depth:

                    connections.discard(tuple(sorted((node, neighbor))))

                min\_back\_depth = min(min\_back\_depth, back\_depth)

            rank[node] = n  # this line is not necessary. see the "brain teaser" section below

            return min\_back\_depth

        dfs(0, 0)  # since this is a connected graph, we don't have to loop over all nodes.

        return list(connections)

**Complexity analysis**

DFS time complexity is O(|E| + |V|), attempting to visit each edge at most twice. (the second attempt will immediately return.)

As the graph is always a connected graph, |E| >= |V|.

So, time complexity = O(|E|).

Space complexity = O(graph) + O(rank) + O(connections) = 3 \* O(|E| + |V|) = O(|E|).

**FAQ: Are you reinventing Tarjan?**

Honestly, I didn't know Tarjan beforehand. The idea of using rank is inspired by preordering which is a basic concept of DFS. Now I realize they are similar, but there are still major differences between them.

This solution uses only one array rank. While Tarjan uses two arrays: dfn and low.

This solution's min\_back\_depth is similar to Tarjan's low, but rank is very different than dfn. max(dfn) is always n-1, while max(rank) could be smaller than n-1.

This solution construsts the result by removing non-critical edges during the dfs, while Tarjan constructs the result by collecting non-critical edges after the dfs.

In this solution, only nodes actively in the current search path have 0<=rank[node]<n; while in Tarjan, nodes not actively in the current search path may still have 0<=dfn[node]<=low[node]<n.

**Brain teaser**

Thanks [@migfulcrum](https://leetcode.com/migfulcrum) for [pointing out](https://leetcode.com/discuss/comment/359567) that rank[node] = n is not necessary. He is totally right. I'll leave this as a brain teaser for you: why is it not necessary?

(Hint: after we've finished visiting a node, is it possible to have another search path attempting to visit this node again?)

**Solution 2: Tarjan's Algorithm (720min)**

**Style 1: Using 'visited' array**

class Solution {

    public List<List<Integer>> criticalConnections(int n, List<List<Integer>> connections) {

        List<List<Integer>> result = new ArrayList<List<Integer>>();

        // Build undirected graph

        Map<Integer, List<Integer>> graph = new HashMap<Integer, List<Integer>>();

        for(int i = 0; i < n; i++) {

            graph.put(i, new ArrayList<Integer>());

        }

        for(List<Integer> connection : connections) {

            int from = connection.get(0);

            int to = connection.get(1);

            graph.get(from).add(to);

            graph.get(to).add(from);

        }

        // Track node's id

        int[] ids = new int[n];

        // Track node's low link (default value is the index)

        int[] lowlinks = new int[n];

        // Track if visit node or not

        boolean[] visited = new boolean[n];

        // Since this is a connected graph, we don't have to loop over all nodes, start with node 0

        //for(int i = 0; i < n; i++) {

        //    if(!visited[i]) {

        //        helper(result, graph, ids, lowlinks, visited, -1, i, 0);

        //    }

        //}

        helper(result, graph, ids, lowlinks, visited, -1, 0, 0);

        return result;

    }

    private void helper(List<List<Integer>> result, Map<Integer, List<Integer>> graph, int[] ids, int[] lowlinks, boolean[] visited, int parent\_node, int cur\_node, int node\_id) {

        ids[cur\_node] = node\_id;

        lowlinks[cur\_node] = node\_id;

        visited[cur\_node] = true;

        //node\_id += 1;

        for(int next\_node : graph.get(cur\_node)) {

            // If encounter parent again, skip

            if(next\_node == parent\_node) {

                continue;

            }

            if(!visited[next\_node]) {

                helper(result, graph, ids, lowlinks, visited, cur\_node, next\_node, node\_id + 1);

                lowlinks[cur\_node] = Math.min(lowlinks[cur\_node], lowlinks[next\_node]);

                // Find the bridge(critical connection)

                if(ids[cur\_node] < lowlinks[next\_node]) {

                    List<Integer> bridge = new ArrayList<Integer>();

                    bridge.add(cur\_node);

                    bridge.add(next\_node);

                    result.add(bridge);

                }

            // next\_node is already visited, cur\_node & next\_node forms a cycle

            // which means tried to visit an already visited node, which may have

            // a lower id than the current low link value

            } else {

                lowlinks[cur\_node] = Math.min(lowlinks[cur\_node], ids[next\_node]);

            }

        }

    }

}

**Refer to**

<https://leetcode.com/problems/critical-connections-in-a-network/discuss/382632/Java-implementation-of-Tarjan-Algorithm-with-explanation>

<https://leetcode.com/problems/critical-connections-in-a-network/discuss/382632/Java-implementation-of-Tarjan-Algorithm-with-explanation/510175>

public List<List<Integer>> criticalConnections(int n, List<List<Integer>> connections) {

        List<List<Integer>> result = new ArrayList<>();

        List<List<Integer>> graph = buildGraph(n, connections);

        int ids[] = new int[n];

        int lowlink[] = new int[n];

        boolean visited[] = new boolean[n];

        for (int i = 0; i < n; i++) {

            if (!visited[i])

                dfs(graph, ids, lowlink, visited, result, i, -1, 0);

        }

        return result;

    }

    private void dfs(List<List<Integer>> graph, int[] ids, int[] lowlink, boolean[] visited, List<List<Integer>> result, int u, int parent, int time) {

        ids[u] = time;

        lowlink[u] = time;

        visited[u] = true;

        for (int v : graph.get(u)) {

            if (v == parent)//if vertex is parent, skip

                continue;

            if (!visited[v]) {

                dfs(graph, ids, lowlink, visited, result, v, u, time + 1);

                lowlink[u] = Math.min(lowlink[u], lowlink[v]);

                if (ids[u] < lowlink[v]) { //critical connections or bridges

                    List<Integer> bridge = new ArrayList<>();

                    bridge.add(u);

                    bridge.add(v);

                    result.add(bridge);

                }

            } else { // v is already traversed. u & v forms a cycle.

                lowlink[u] = Math.min(lowlink[u], ids[v]);

            }

        }

    }

    private List<List<Integer>> buildGraph(int n, List<List<Integer>> connections) {

        List<List<Integer>> graph = new ArrayList<>();

        for (int i = 0; i < n; i++) {//add vertices

            graph.add(new ArrayList<>());

        }

        for (List<Integer> edge : connections) { //add edges

            int from = edge.get(0);

            int to = edge.get(1);

            graph.get(from).add(to);

            graph.get(to).add(from);

        }

        return graph;

    }

**Style 2: Not using 'visited' array**

class Solution {

    public List<List<Integer>> criticalConnections(int n, List<List<Integer>> connections) {

        List<List<Integer>> result = new ArrayList<List<Integer>>();

        // Build undirected graph

        Map<Integer, List<Integer>> graph = new HashMap<Integer, List<Integer>>();

        for(int i = 0; i < n; i++) {

            graph.put(i, new ArrayList<Integer>());

        }

        for(List<Integer> connection : connections) {

            int from = connection.get(0);

            int to = connection.get(1);

            graph.get(from).add(to);

            graph.get(to).add(from);

        }

        // Track node's id

        int[] ids = new int[n];

        // Track node's low link (default value is the index)

        int[] lowlinks = new int[n];

        // No need 'visited' array to track the node visited or not

        // since initially we can assign all node's id as -1, when

        // DFS traversal, if visited a node, we update the node id,

        // which means if the node id keep as -1, its not visited

        //boolean[] visited = new boolean[n];

        Arrays.fill(ids, -1);

        // Since this is a connected graph, we don't have to loop over all nodes, start with node 0

        //for(int i = 0; i < n; i++) {

        //    if(!visited[i]) {

        //        helper(result, graph, ids, lowlinks, visited, -1, i, 0);

        //    }

        //}

        helper(result, graph, ids, lowlinks, -1, 0, 0);

        return result;

    }

    private void helper(List<List<Integer>> result, Map<Integer, List<Integer>> graph, int[] ids, int[] lowlinks, int parent\_node, int cur\_node, int node\_id) {

        ids[cur\_node] = node\_id;

        lowlinks[cur\_node] = node\_id;

        //visited[cur\_node] = true;

        for(int next\_node : graph.get(cur\_node)) {

            // If encounter parent again, skip

            if(next\_node == parent\_node) {

                continue;

            }

            // If the node id keep as -1, its not visited

            if(ids[next\_node] == -1) {

                helper(result, graph, ids, lowlinks, cur\_node, next\_node, node\_id + 1);

                lowlinks[cur\_node] = Math.min(lowlinks[cur\_node], lowlinks[next\_node]);

                // Find the bridge(critical connection)

                if(ids[cur\_node] < lowlinks[next\_node]) {

                    result.add(Arrays.asList(cur\_node, next\_node));

                }

            // next\_node is already visited, cur\_node & next\_node forms a cycle

            // which means tried to visit an already visited node, which may have

            // a lower id than the current low link value

            } else {

                lowlinks[cur\_node] = Math.min(lowlinks[cur\_node], ids[next\_node]);

            }

        }

    }

}

**Style 3: Not using 'visited' array and no need HashMap**

class Solution {

    public List<List<Integer>> criticalConnections(int n, List<List<Integer>> connections) {

        List<List<Integer>> result = new ArrayList<List<Integer>>();

        List<List<Integer>> graph = new ArrayList<>();

        for(int i = 0; i < n; i++) {

            graph.add(new ArrayList<Integer>());

        }

        for(List<Integer> connection : connections) {

            int from = connection.get(0);

            int to = connection.get(1);

            graph.get(from).add(to);

            graph.get(to).add(from);

        }

        int[] ids = new int[n];

        int[] lowlinks = new int[n];

        Arrays.fill(ids, -1);

        helper(result, graph, ids, lowlinks, -1, 0, 0);

        return result;

    }

    private void helper(List<List<Integer>> result, List<List<Integer>> graph, int[] ids, int[] lowlinks, int parent\_node, int cur\_node, int node\_id) {

        ids[cur\_node] = node\_id;

        lowlinks[cur\_node] = node\_id;

        for(int next\_node : graph.get(cur\_node)) {

            if(next\_node == parent\_node) {

                continue;

            }

            if(ids[next\_node] == -1) {

                helper(result, graph, ids, lowlinks, cur\_node, next\_node, node\_id + 1);

                lowlinks[cur\_node] = Math.min(lowlinks[cur\_node], lowlinks[next\_node]);

                if(ids[cur\_node] < lowlinks[next\_node]) {

                    result.add(Arrays.asList(cur\_node, next\_node));

                }

            } else {

                lowlinks[cur\_node] = Math.min(lowlinks[cur\_node], ids[next\_node]);

            }

        }

    }

}

**Refer to**

[Tarjan's Algorithm Strongly Connected Components](note://BAE78B183ABA4840ACA2F00EEDDA774D)

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])

**Java version**

class Solution {

    private int time = 0; // Tracks discovery time

    private List<List<Integer>> bridges = new ArrayList<>(); // Stores the critical connections

    public List<List<Integer>> criticalConnections(int n, List<List<Integer>> connections) {

        // Build the graph as an adjacency list

        List<List<Integer>> graph = buildGraph(n, connections);

        // Initialize arrays to track visited nodes, discovery times, and low-link values

        int[] discovery = new int[n];

        int[] low = new int[n];

        Arrays.fill(discovery, -1); // Use -1 to indicate unvisited nodes

        // Perform DFS starting from node 0

        dfs(0, -1, graph, discovery, low);

        return bridges;

    }

    private void dfs(int node, int parent, List<List<Integer>> graph, int[] discovery, int[] low) {

        discovery[node] = low[node] = time++; // Set discovery and low-link value to the current time

        for (int neighbor : graph.get(node)) {

            if (neighbor == parent) {

                // Skip the edge to the parent node

                continue;

            }

            if (discovery[neighbor] == -1) {

                // If the neighbor is not visited, recursively perform DFS

                dfs(neighbor, node, graph, discovery, low);

                // Update the low-link value of the current node

                low[node] = Math.min(low[node], low[neighbor]);

                // Check if the edge is a bridge

                if (low[neighbor] > discovery[node]) {

                    bridges.add(Arrays.asList(node, neighbor));

                }

            } else {

                // Update the low-link value for a back edge

                low[node] = Math.min(low[node], discovery[neighbor]);

            }

        }

    }

    private List<List<Integer>> buildGraph(int n, List<List<Integer>> connections) {

        // Create an adjacency list

        List<List<Integer>> graph = new ArrayList<>();

        for (int i = 0; i < n; i++) {

            graph.add(new ArrayList<>());

        }

        for (List<Integer> connection : connections) {

            int u = connection.get(0);

            int v = connection.get(1);

            graph.get(u).add(v);

            graph.get(v).add(u);

        }

        return graph;

    }

}

**Explanation of Conversion:**

Graph Construction:

A List<List<Integer>> is used in Java to represent the adjacency list, similar to defaultdict(list) in Python.

Tracking Discovery and Low-link Values:

The discovery and low arrays are initialized with -1 to indicate unvisited nodes, as Java does not allow default array initialization with custom values like Python.

DFS Implementation:

The recursive DFS function uses the same logic as the Python version:

Update the discovery and low values for the current node.

Recursively visit all neighbors except the parent node.

After recursion, update the low-link values and check for bridges.

Bridges:

If low[neighbor] > discovery[node], the edge (node, neighbor) is a critical connection and is added to the bridges list.

Output:

The bridges list contains all critical connections, equivalent to the Python bridges list.

**Complexity:**

Time Complexity: O(V+E), where V is the number of nodes and E is the number of edges. The graph is traversed once during DFS.

Space Complexity: O(V+E) for the graph representation and recursive call stack.

[Video explain for Tarjan's Algorithm](https://www.youtube.com/watch?v=aZXi1unBdJA)

<https://www.youtube.com/watch?v=aZXi1unBdJA>

**Refer to**

[Tarjan's Algorithm Strongly Connected Components](note://BAE78B183ABA4840ACA2F00EEDDA774D)

[Directed and Undirected Graph Cycle Detection in DFS and BFS](note://8D51E6EB87834766A8F4688C191B70AC)

[Directed and Undirected Graph Connectedness](note://703FB1A74E454327B7F34E47F95EDAB5)

[Graph Traversals and Directed Graph Cycle Detection](note://78C8A0548F4140BDB1F12AEF3491A0A1)