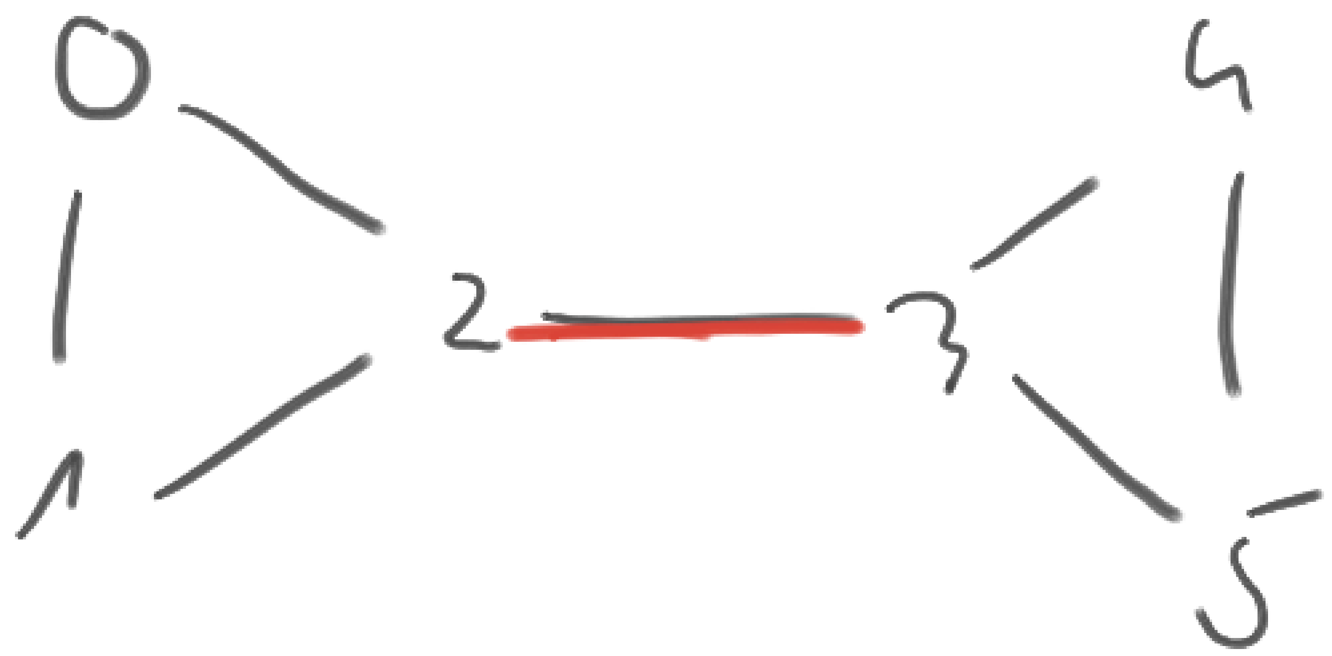
Tutorium #5

25/11/2022

Given a connected undirected graph G = (V, E) (without self loops), return all bridges

An edge is a bridge iff it's removal increases the number of SCC (which is the same to CC, since undirected)

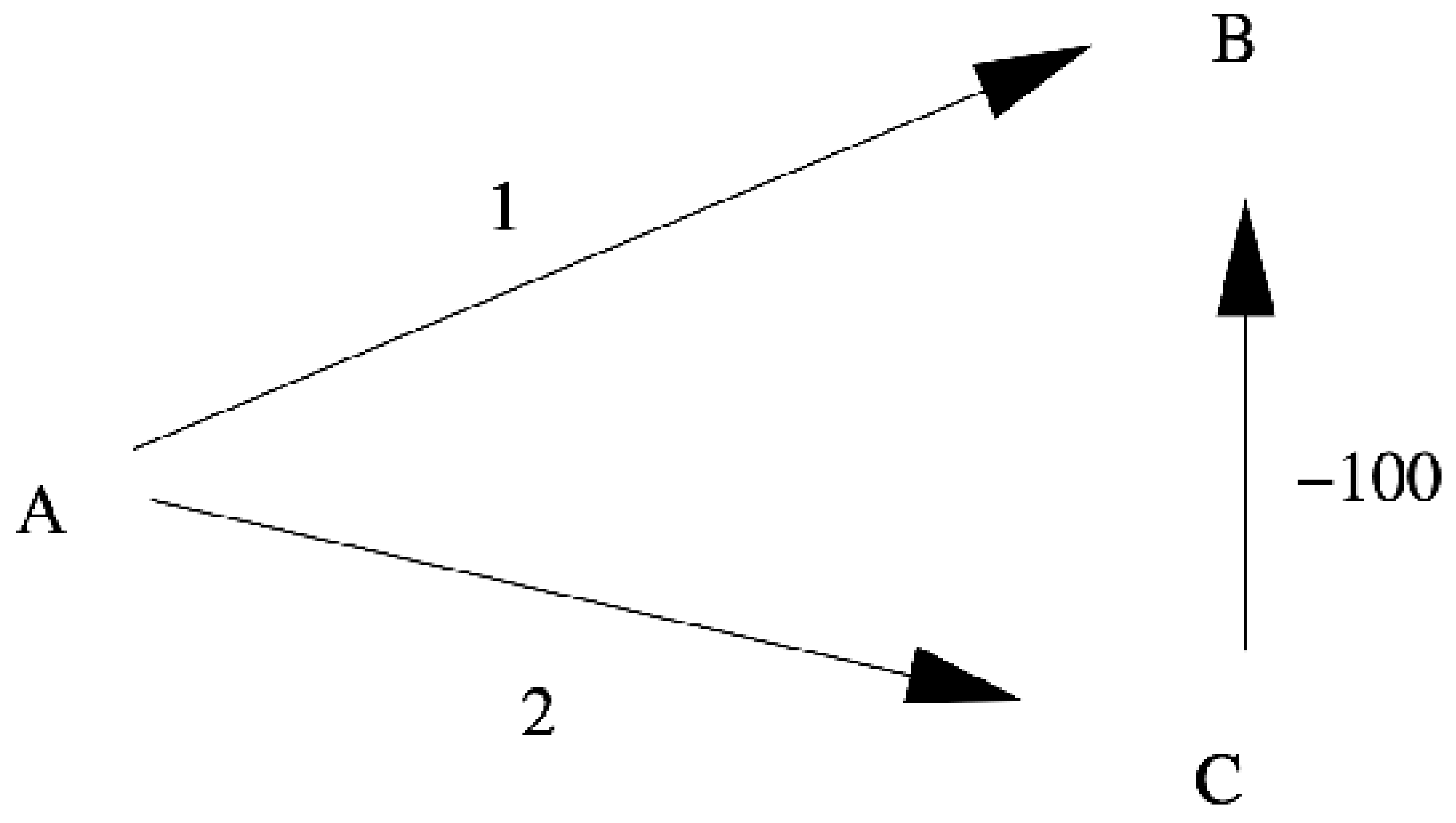


$e \in E$ is bridge $\Leftrightarrow$ it doesn't belong to a cycle

```
def findBridges(graph):
  lowestReachableVertex = [INFTY * graph.numberOfNodes]
  bridges = []

  // start with first vertex 0, and discoveryTime 0
  DFS(0, 0, graph)

  return bridges

def DFS(node, time, graph):
  if (lowestReachableVertex[node] == INFTY):
    lowestReachableVertex[node] = time
    for neighbor in graph[node]:
      lowestReachableNode = DFS(neighbor, time + 1, graph)
      // if neighbor finds a ''larger'' vertex, this edge doesn't belong to a cycle => bridge
      if (lowestReachableNode > time):
        bridges.append([node, neighbor])
      lowestReachableVertex[node] = min(lowestReachableVertex[node], lowestReachableNode)
  return lowestReachableVertex[node]
```

Problem 1 (2 points) Give an example graph with edge costs and a start node that shows that Dijkstra may not compute shortest paths correctly if negative edge costs are allowed.

Solution:
In the following network pick $A = s$. Dijkstra relaxes the edges (A, B) and (A, C). Afterwards the preliminary distance of B is 1 and of C is 2. Hence, B will be the next node that is settled. This is a problem as Dijkstra has the property that settled nodes have an optimal distance label. This is however not the case here since the path $A \rightsquigarrow C \rightsquigarrow B$ gives us a shorter distance.



Note

- your graph should have no neg. cycle

- directed

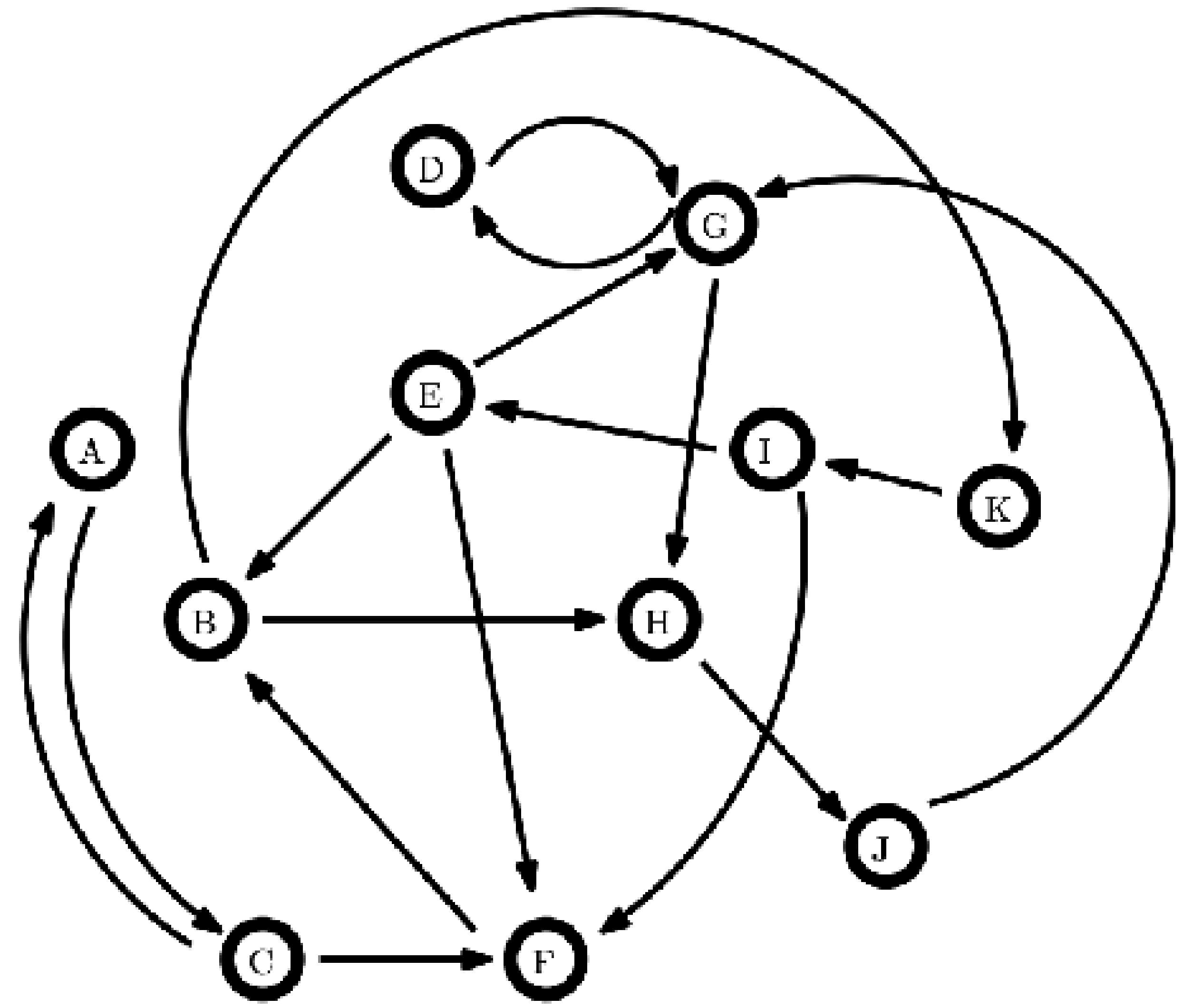
Problem 2 (6 points)

Given is a directed graph $G = (V, E)$ and the following relation

$$v \overset{*}{\longleftrightarrow} w \qquad \text{iff.} \qquad \text{exist paths } \langle v, \ldots, w \rangle \text{ and } \langle w, \ldots, v \rangle \text{ in } G$$

for $v, w \in V$.

1. Show $\overset{*}{\longleftrightarrow}$ is an equivalence relationship.
2. Mark **all** *strongly connected components* in the following graph.

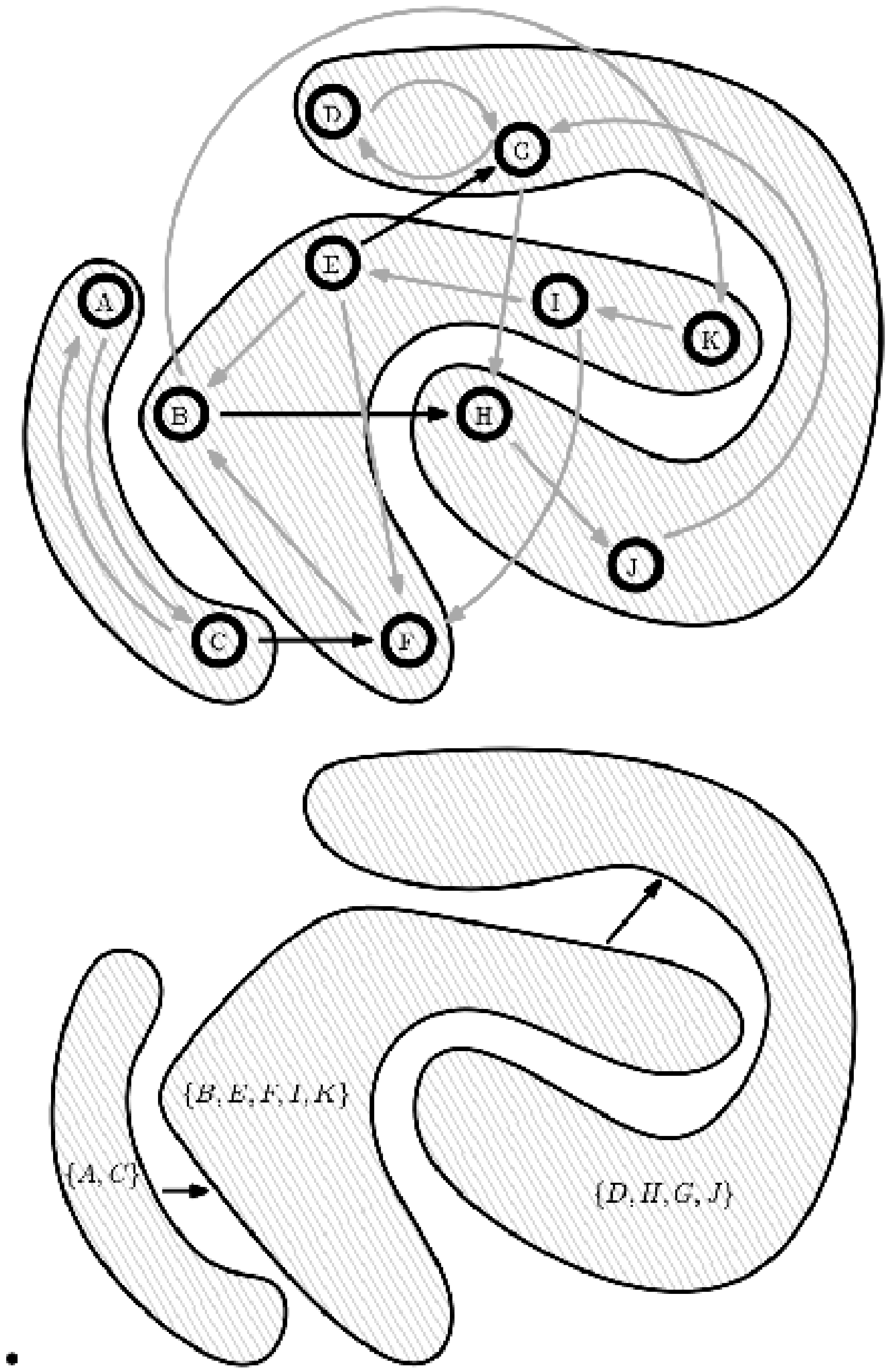


- **Reflexivity:** $(\forall v \in V : v \leftrightarrow v)$
  There is a trivial path from v to v for each vertex.

  Symmetry: $(v \leftrightarrow w \implies w \leftrightarrow v)$
  $v \leftrightarrow w$:
  $\exists$ path from v to w and from w to v in G:
  $\exists$ path from w to v and from v to w in G:
  $w \leftrightarrow v$

  Transitivity: $(v \leftrightarrow w \text{ and } w \leftrightarrow x \implies v \leftrightarrow x)$
  If $v \leftrightarrow w$ and $w \leftrightarrow x$, there exist paths from v to w, from w to x, from x to w and from w to v. Therefore there exists a path form v to x consisting of the path from v to w followed by the path from w to x. The reverse path also exists as the concatentation of the reverse paths, which also exist due to our assumptions. Therefore $v \leftrightarrow x$ and $\leftrightarrow$ is transitive.

  $\leftrightarrow$ is reflexive, symmetric and transitive and therefore and equivalence relation.


A
B
C
D
E
F
G
H
I
J
K
$\{A, C\}$
$\{B, E, F, I, K\}$
$\{D, H, G, J\}$

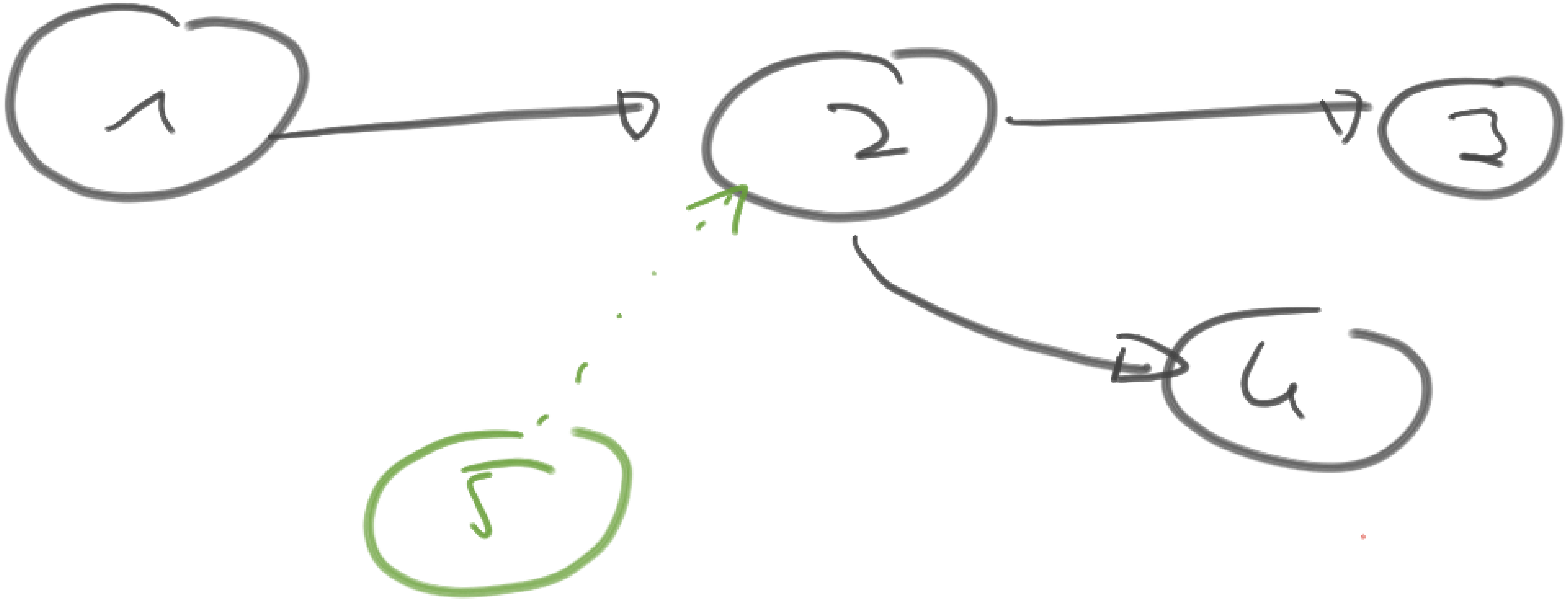
3. Compute and draw the SCC contracted graph.
4. Prove that the SCC contracted graph of a graph is acyclic.
5. Given is a directed, not necessarily connected graph $G = (V, E)$. Design a linear time algorithm that outputs all vertices from which all other vertices can be reached. (Note: this part of the exercise gives 2 points)

- For contradiction, assume there exists a cycle $n_1,\ n_2,\ \ldots\ n_j = n_1$ with $j > 1$. By definition each of the vertices in the contracted graph is a maximal SCC in the original graph. As the component graph contains a cycle by assumption, there exists an edge for every component $m \in [1, \ldots, j]$ from at least one vertex $w_{o,m}$ in n_m to a vertex $w_{i,m+1}$ in n_{m+1}. Per definition of the SCC, there also exists a path from $w_{i,m}$ to $w_{o,m}$ in each component.

  Let u be a node in component n_k and v be a node in a different component n_l (w.l.o.g $k < l$). There exists a path from u over $w_{o,k}$, $w_{i,k+1}$,...$w_{o,l-1}$, $w_{i,l}$ to v. Following the same argument there exists a path from v to u. Therefore nodes u and v are in the same SCC, which results in a contradiction to the definition of n_k and n_l being maximal strongly connected components.

- First, use the algorithm from the lecture to compute the SCCs in linear time. The previous exercise shows that the contracted graph is a directed acyclic graph (DAG).

  If there is only one node with in-degree 0, this node is the root and all nodes which are represented by this contracted node will be output. If there are multiple nodes with in-degree 0, no node in the graph can reach all other nodes and no node will be output.



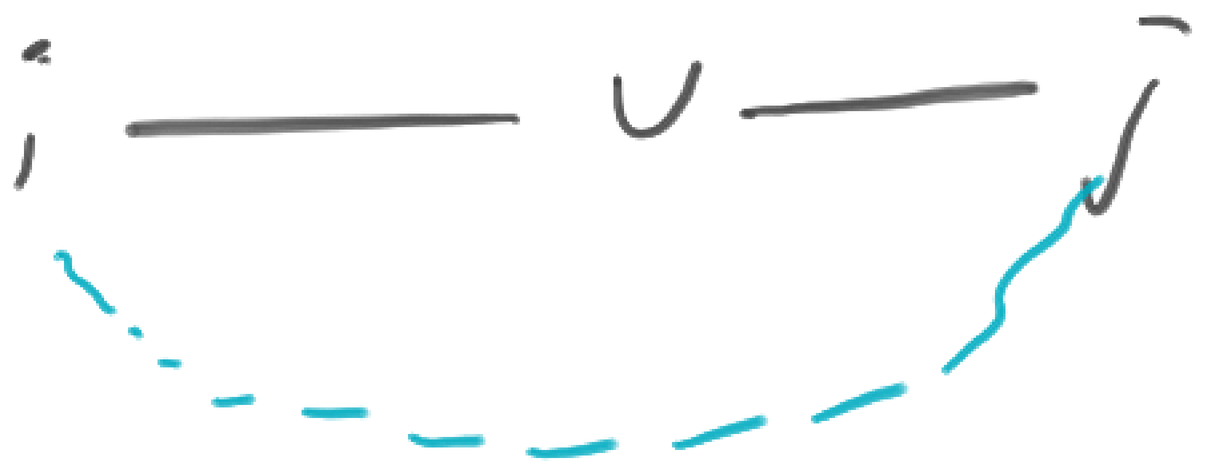
Topological sorting idea

Let $G = (V, E)$ be a connected undirected graph. A node v is called *articulation point*, if the removal of the node increases the number of connected components.

1. Show that in a graph *without articulation points* and $|V| \geq 3$ there is always at least one pair of nodes (i, j), $i, j \in V$, so that there are two paths $P_1 = \langle i, \ldots, j \rangle$ and $P_2 = \langle i, \ldots, j \rangle$, that are node disjoint expect for the endpoints, i.e.: $P_1 \cap P_2 = \{i, j\}$.

Solution:

1. Let v be not an articulation point and i and j two v of v. On path of i and j is obviously $P_1 = \langle (i, v), (v, j) \rangle$. If we remove v, those edges are removed as well. G stays connected, otherwise v would be an articulation point. Hence, there must be a second P_2 between i and j that is also disjoint to P_1, since v is not contained in the graph anymore.



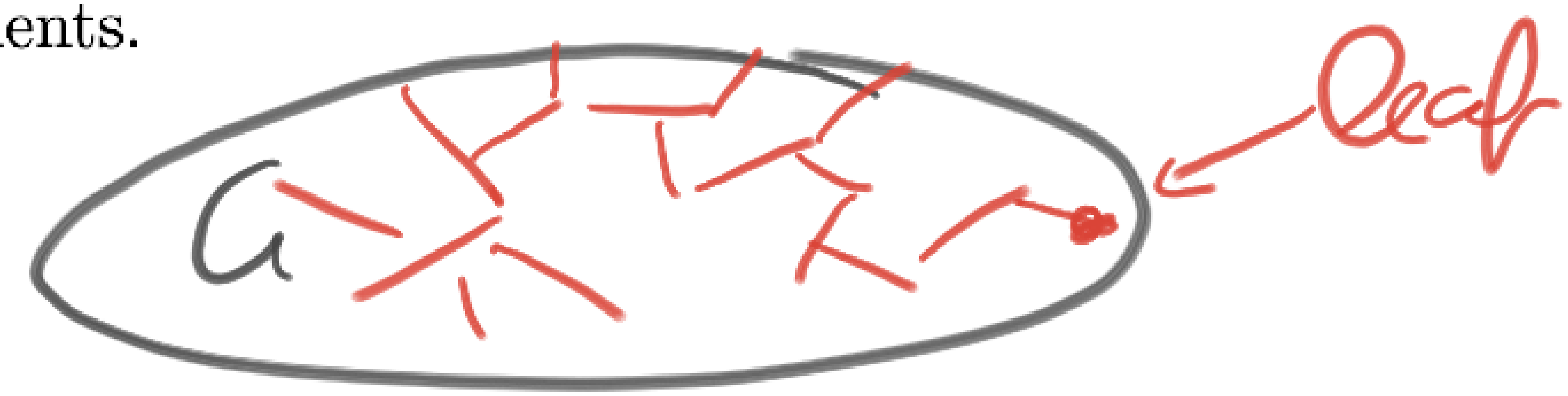
2. Show that in a graph with *articulation ponts* there is a node v, such that: You can remove a node w, such that starting from v there are no more paths to at least half of the remaining node.

3. Show, in a connected graph $G = (V, E)$ there is always a node v, such that G is connected after v is removed.

2. Let w be an articulation point. Then G has two or more components after removal of v. One component K has the smallest number of nodes among all components. Since there are at least two components and K is the smaller one, we get that K can not contain more than $|K| := \frac{|V \setminus \{v\}|}{2}$ nodes. If you choose a node v from K, then this node can obivously reach less than half of the remaining nodes.

3. Consider a spanning tree of G. Every leaf of the tree can be removed without increasing the number of connected components.

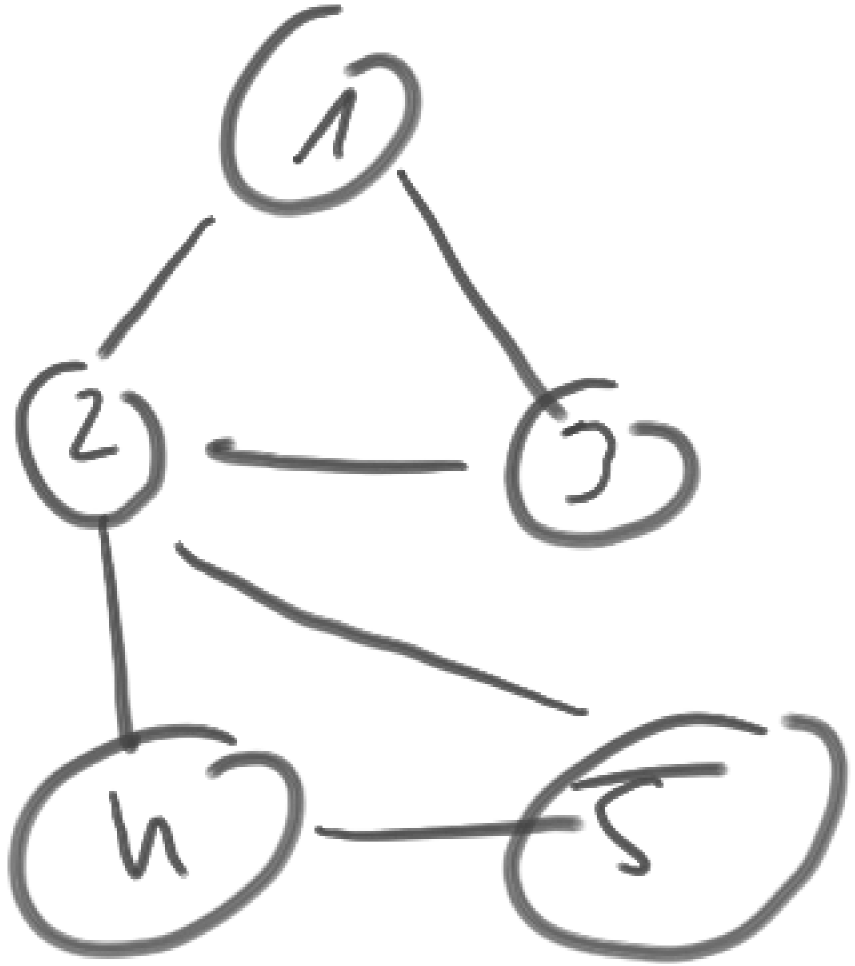
spanning tree

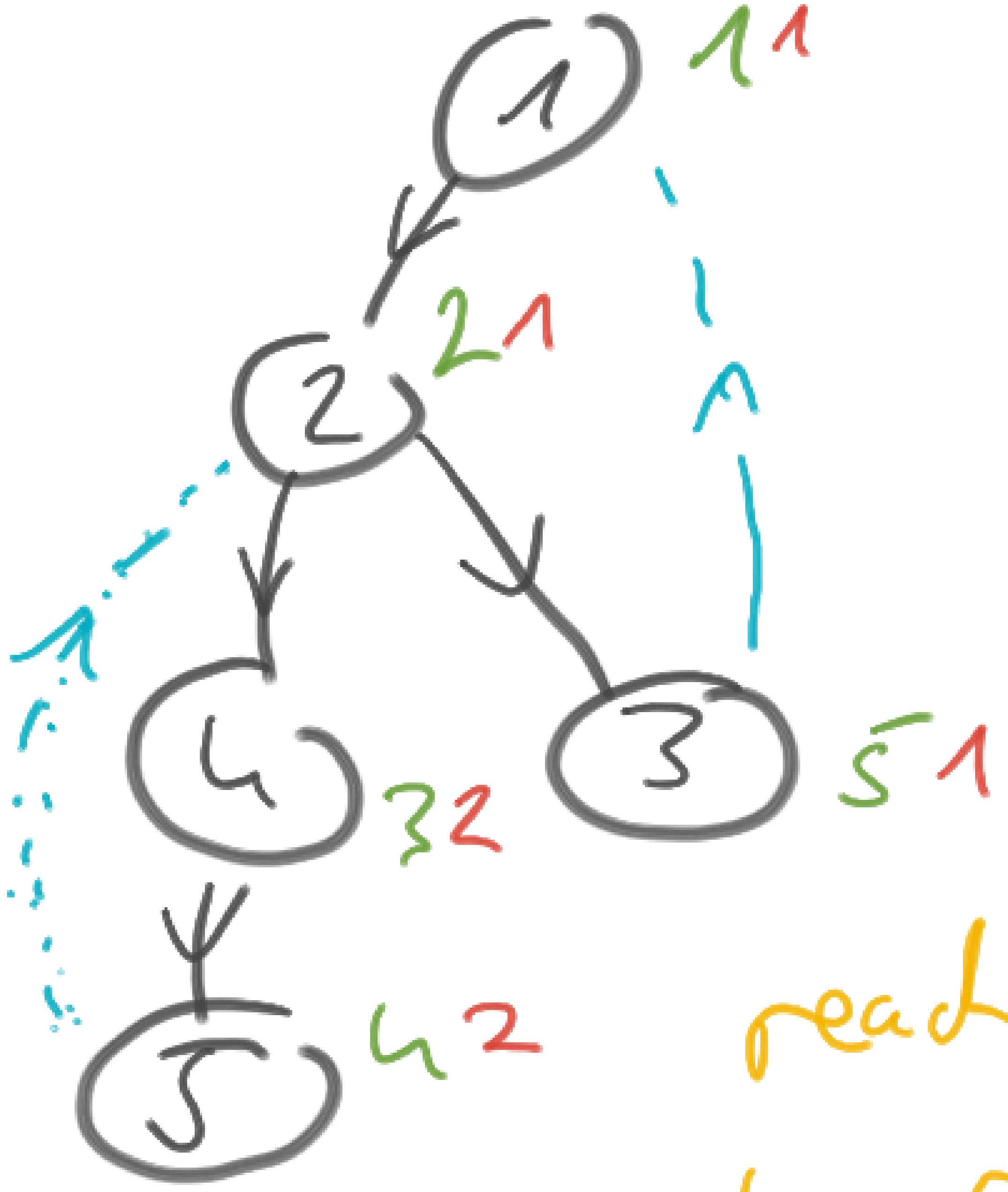


4. Complete the following DFS-Algorithm, such that in time $O(|V|+|E|)$ it output all articulation points in a undirected graph. Describe what the functions `init`, `root(s)`, `traverseTreeEdge(v,w)`, `traverseNonTreeEdge(v,w)` and `backTrack(u,v)` do.
First think about how you can identify articulation points via the DFS-numbering.

```
Depth-first search of graph G = (V, E)
unmark all nodes
init
for all s ∈ V do
    if s is not marked then
        mark s
        root(s)
        DFS(s,s)

procedure DFS(u,v : NodeID)
    for all (v, w) ∈ E do
        if w is marked then
            traverseNonTreeEdge(v,w)
        else
            traverseTreeEdge(v,w)
            mark w
            DFS(v,w)
    backtrack(u,v)
```

The problem can be solved using DFS. The following observation yields the key to the solution: A node v is always an articulation point, if he cannot reach a node that has a smaller DFS number. To check this, the minimum number of reachable DFS numbers of all subtrees have to be propagated upwards. The root is a special case and only articulation point, if it contains multiple successors.





dfsNum

minReachable

when backtracking (2,4), we see that 2 can reach "deeper" than 4 ⇒ 2 is an articulation point

init:	dfsPos= 1; finishingTime= **1**
root(s):	dfsNum[s]=dfsPos++; minimum[s] = dfsNum[s]; tree_root = s
traverseTreeEdge(v,w):	dfsNum[w]:=dfsPos++; minimum[w] = dfsnum[w]
traverseNonTreeEdge(v,w):	minimum[v] = min(dfsNum[w], minimum[v])
backtrack(u,v):	minimum[u] = min(minimum[u], minimum[v])

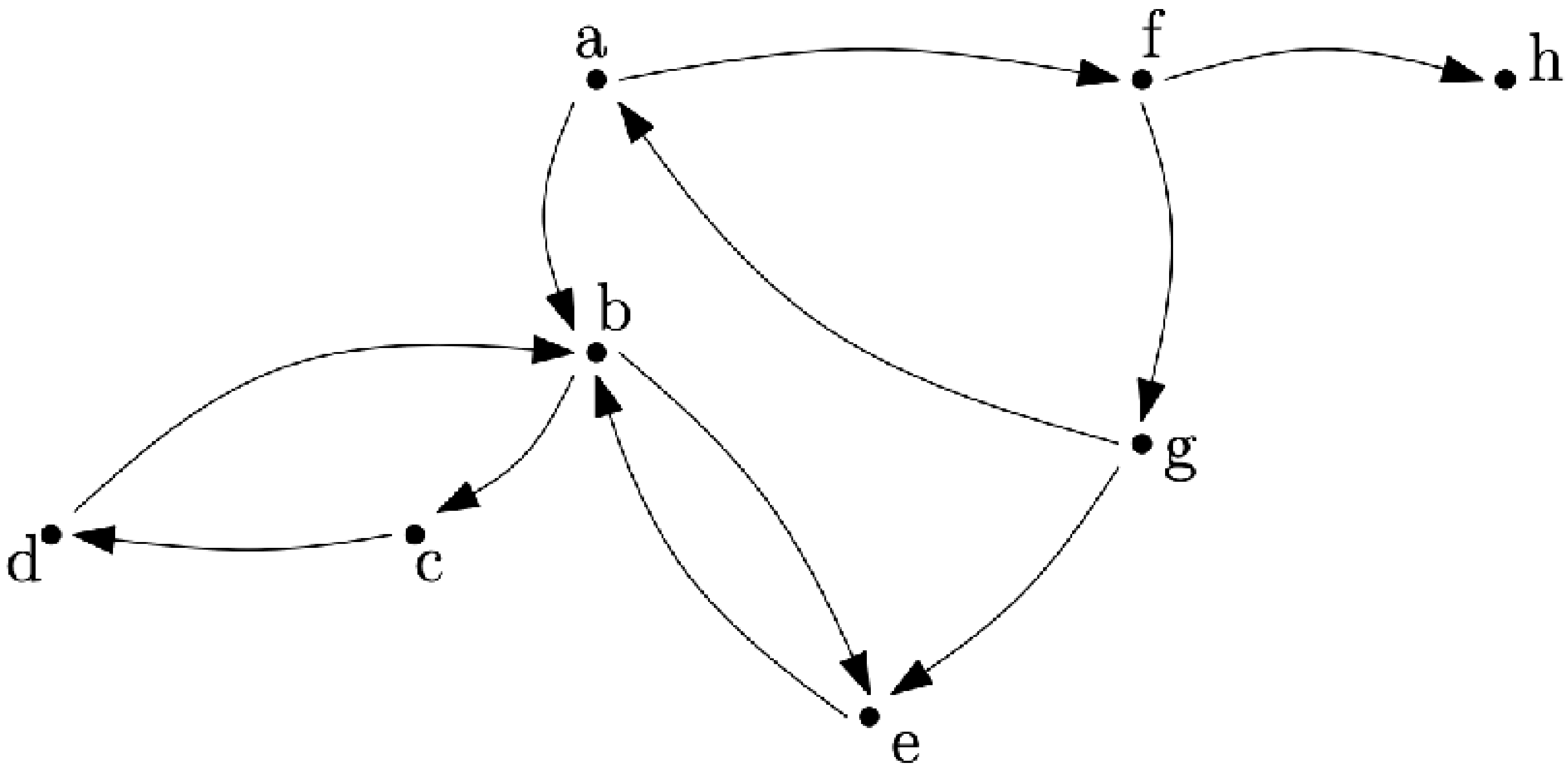
```
if( minimum[v] ≥ dfsNum[u] &&
    ( tree_root ≠ u || # childs(u) > 1 ) )
      output(u)
```

```
def articulationPoints(graph):
  seen = [False * graph.numberOfNodes]
  dfsNumber = [-1 * graph.numberOfNodes]
  minReachable = [INFTY * graph.numberOfNodes]
  rootVertex = -1
  // init
  dfsCounter = 1

  for v in graph.nodes():
    if not seen[v]:
      seen[v] = True
      dfsNumber[v] = dfsCounter++
      minReachable[v] = dfsNumber[v]
      rootVertex = v
      DFS(v, v, graph)
```

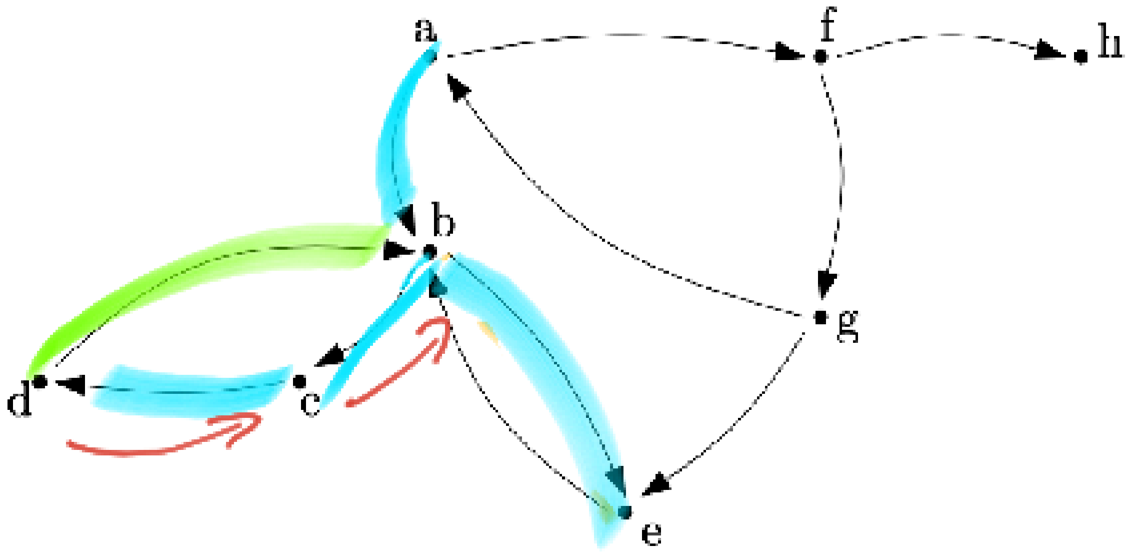
```
def DFS(u, v, graph):
  for edge in graph[v]:
    if (seen[edge.toVertex]):
      // traverseNonTreeEdge
      minReachable[v] = min(minReachable[v],
dfsNumber[edge.toVertex])
    else:
      // traverseTreeEdge
      dfsNum[edge.toVertex] = dfsCounter++
      minReachable[edge.toVertex] = dfsNumber[edge.toVertex]

      seen[edge.toVertex] = True
      DFS(v, edge.toVertex, graph)

  // backtrack
  minReachable[u] = min(minReachable[u], minReachable[v])
  // check if we found an articulation point
  if (u == rootVertex and len(graph[u]) > 1):
    print("Articulation Point (root): " + str(u))
  if (minReachable[v] >= dfsNumber[u] and u != rootVertex):
    print("Articulation Point (not root): " + str(u))
```

Problem 4 (4 points)

Given is the following graph $G = (V, E)$:



Exectute the algorithm to compute SCCs from the lecture on the graph.
After each step give the state of `oReps`, `oNodes` and `component`.



Schritt 1: `root(a)`

oReps	a
oNodes	a

w	a	b	c	d	e	f	g	h
component[w]	-	-	-	-	-	-	-	-

Schritt 2: `traverseTreeEdge(a,b)`

oReps	b a
oNodes	b a

w	a	b	c	d	e	f	g	h
component[w]	-	-	-	-	-	-	-	-

Schritt 3: `traverseTreeEdge(b,c)`

oReps	c b a
oNodes	c b a

w	a	b	c	d	e	f	g	h
component[w]	-	-	-	-	-	-	-	-

Schritt 4: `traverseTreeEdge(c,d)`

oReps	d c b a
oNodes	d c b a

w	a	b	c	d	e	f	g	h
component[w]	-	-	-	-	-	-	-	-

Schritt 5: `traverseNonTreeEdge(d,b)`

oReps	b a
oNodes	d c b a

w	a	b	c	d	e	f	g	h
component[w]	-	-	-	-	-	-	-	-

Schritt 6, 7: `backtrack(c,d), backtrack(b,c)`

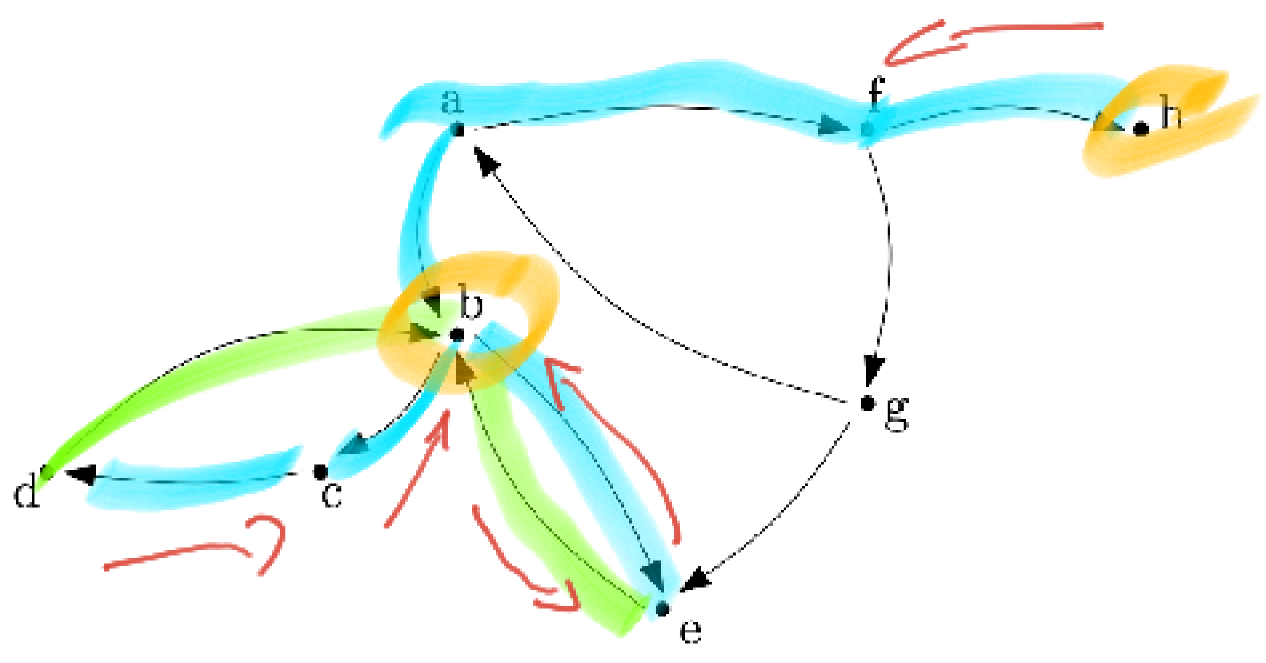
oReps	b a
oNodes	d c b a

w	a	b	c	d	e	f	g	h
component[w]	-	-	-	-	-	-	-	-

Schritt 8: `traverseTreeEdge(b,e)`

oReps	e b a
oNodes	e d c b a

w	a	b	c	d	e	f	g	h
component[w]	-	-	-	-	-	-	-	-



Schritt 9: `traverseNonTreeEdge(e,b)`

oReps	b a
oNodes	e d c b a

w	a	b	c	d	e	f	g	h
component[w]	-	-	-	-	-	-	-	-

Schritt 10: `backTrack(b,e)`

oReps	b a
oNodes	e d c b a

w	a	b	c	d	e	f	g	h
component[w]	-	-	-	-	-	-	-	-

Schritt 11: `backTrack(a,b)`

b is finished

oReps	a
oNodes	a

w	a	b	c	d	e	f	g	h
component[w]	-	b	b	b	b	-	-	-

Schritt 12: `traverseTreeEdge(a,f)`

oReps	f a
oNodes	f a

w	a	b	c	d	e	f	g	h
component[w]	-	b	b	b	b	-	-	-

Schritt 13: `traverseTreeEdge(f,h)`

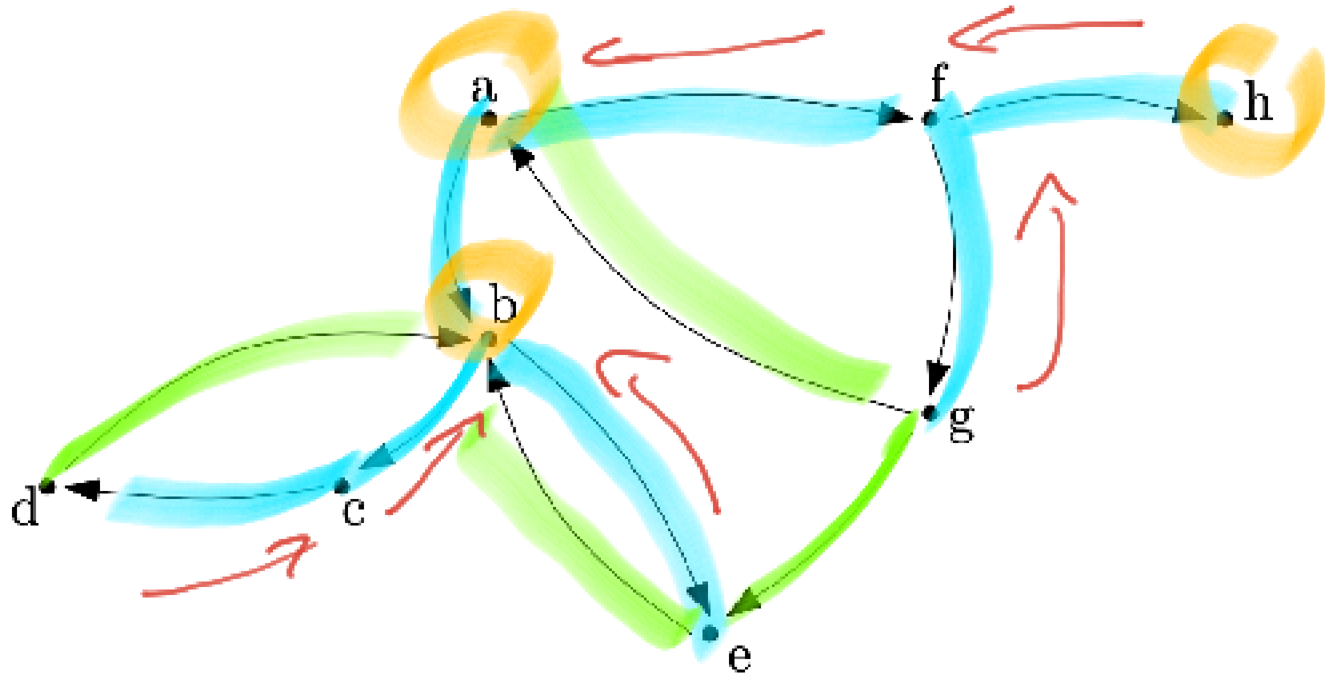
oReps	h f a
oNodes	h f a

w	a	b	c	d	e	f	g	h
component[w]	-	b	b	b	b	-	-	-

Schritt 14: `backtrack(f,h)`

oReps	f a
oNodes	f a

w	a	b	c	d	e	f	g	h
component[w]	-	b	b	b	b	-	-	h



Schritt 15: `traverseTreeEdge(f,g)`

oReps	g f a
oNodes	g f a

w	a	b	c	d	e	f	g	h
component[w]	-	b	b	b	b	-	-	h

Schritt 16: `traverseNonTreeEdge(g,e)`

oReps	g f a
oNodes	g f a

w	a	b	c	d	e	f	g	h
component[w]	-	b	b	b	b	-	-	h

Schritt 17: `traverseNonTreeEdge(g,a)`

oReps	a
oNodes	g f a

w	a	b	c	d	e	f	g	h
component[w]	-	b	b	b	b	-	-	h

Schritt 18: `backtrack(f,g)`, `backtrack(a,f)`

oReps	a
oNodes	g f a

w	a	b	c	d	e	f	g	h
component[w]	-	b	b	b	b	-	-	h

Schritt 19: `backtrack(a,a)`

oReps	
oNodes	

w	a	b	c	d	e	f	g	h
component[w]	a	b	b	b	b	a	a	h