Network Algorithms

Chapter 5
Network Resilience

Introduction and Motivation

- **Network resilience** denotes the property of a network to sustain the ability to communicate even if parts (nodes, links) of the network fail
  - Failures can occur **by random** or because of **deliberate attacks**
  - Random failures often have less severe consequences and are thus easier to account for
- We are thus often interested in **quantifying the resilience** of a network to random or intentional failures
  - Quantification of random failures often computes the probability of certain conditions, e.g. partitioning of a network etc.
  - Quantification of failures due to deliberate attacks often computes the worst case damage, e.g. smallest number of attacked/failed links or nodes so that the remaining network is partitioned etc.
- Likewise, we are interested in **computing the smallest number of additional links (or nodes)** that need to be added in order to increase the resilience of a network against random failures or deliberate attacks

Some Definitions From Graph Theory (1)

- Two paths p_1 and p_2 from x to y in G are **edge independent** if they have no link in common
- Two paths p_1 and p_2 from x to y in G are **node independent** if they only have nodes x and y in common
- If there is at least one path linking every pair of actors in the graph then the graph is called **connected**
- If there are k edge-independent paths connecting every pair, the graph is **k-edge-connected**
- If there are k node-independent paths connecting every pair, the graph is **k-node-connected**
- The biggest number k for which G is k-edge-connected is called the **edge-connectivity of G**
- The biggest number k for which G is k-node-connected is called the **node-connectivity of G**
  - Graphs that are 2-node-connected are also called **biconnected**

Some Definitions From Graph Theory (2)

- In any connected component, the path(s) linking two non-adjacent nodes must pass through a subset of other nodes, which if removed, would disconnect them
  - For two nodes s and t the set $T \subseteq (V \setminus \{s, t\})$ is called an **s-t-cutting-node-set** if every path connecting s and t passes through at least one node of T, that is there is not path from s to t in $G \setminus T$
  - A set T is called a **cutting-node-set** if T is an s-t-cutting-node-set for two nodes s and t
  - For two nodes s and t the set $F \subseteq E$ is called an **s-t-cutting-edge-set** if every path connecting s and t traverses at least one edge of F, that is there is not path from s to t in $G \setminus F$
  - A set F is called a **cutting-edge-set** if F is an s-t-cutting-edge-set for two nodes s and t

Menger's Theorem

- **Menger's Theorem (1927):**
  - For non-adjacent nodes s and t in an undirected graph, the maximum number of node independent paths is equal to the minimum size of an s-t-cutting-node-set
  - For nodes s and t in an undirected graph, the maximum number of edge independent paths is equal to the minimum size of an s-t-cutting-edge-set

- Menger's Theorem can be interpreted as an early version of the Max-Flow-Min-Cut-Theorem of Ford and Fulkerson, with the help of which it can be easily proven

- Menger's Theorem further allows to obtain the following interesting result

Whitney's Theorem

- **Whitney's Theorem (1932):**
  - An undirected graph with at least k+1 nodes is k-node-connected if and only if each cutting-node-set in G contains at least k nodes
  - An undirected graph is k-edge-connected if and only if each cutting-edge-set in G contains at least k edges

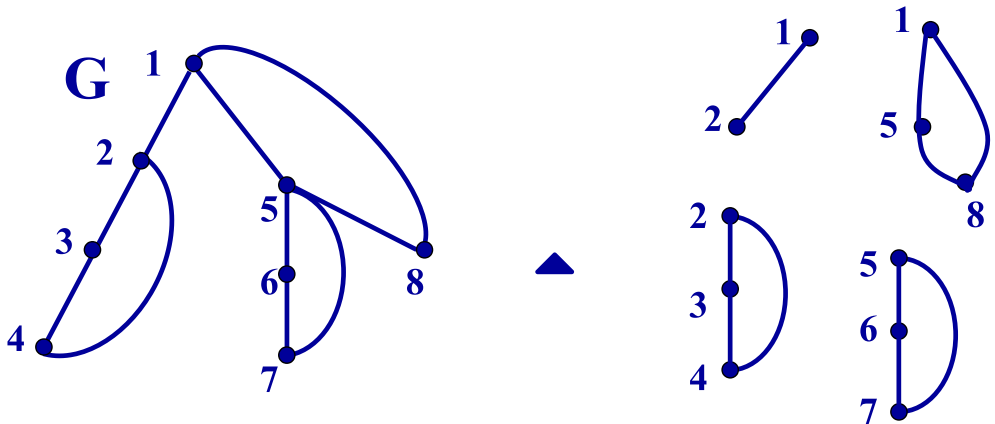
- Implications for communication networks:
  - If a communication network is supposed to allow communication between arbitrary nodes even in case of failure of r arbitrary nodes, its topology must be at least (r+1)-node-connected
  - If a communication network is supposed to allow communication between arbitrary nodes even in case of failure of s arbitrary links, its topology must be at least (s+1)-edge-connected

Interesting Problems Arising From This

- From an algorithmic point of view, this motivates the interest in the following problems:
  - Check for a given graph G and a given number k if G is k-node-connected and/or k-edge-connected (can be solved in polynomial time)
  - Compute for a given graph G the largest number k for which G is k-node-connected and/or k-edge-connected (can be solved in polynomial time)
  - Augment a given graph G that is not k-node-connected or k-edge-connected with the minimum set of edges with which the graph can be made k-node-connected or k-edge-connected
    - For edge-connectivity this can be solved in polynomial time
    - For node-connectivity polynomial algorithms are only known for rather small numbers k ($k \leq 4$).
    - The weighted variant with the objective of weight minimization is NP-hard already for k=2

Block Structure of Graphs (1)

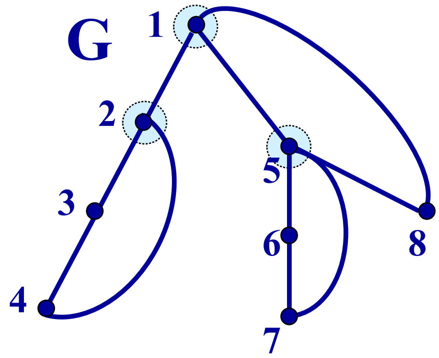
- In the following, we consider undirected graphs G = (V, E) with at least 3 nodes, and we are looking for all subgraphs of maximum size that are (at least) 2-node-connected (biconnected)
- Recall: G is biconnected if and only if (iff) either
  - G is a single edge, or
  - for each tuple of vertices u, v there are at least two node disjoint paths



Biconnected Components of a Graph

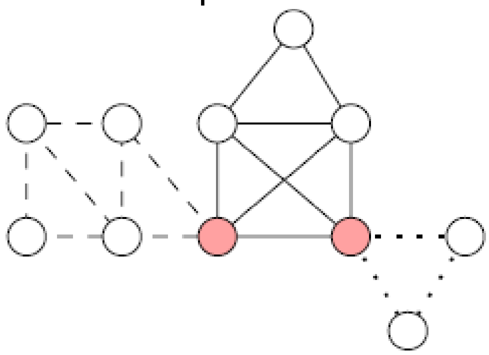
Biconnected Components Meet at Articulation Nodes

- The intersection of two maximum size biconnected components consists of at most one vertex called an **articulation node**
  - Example: 1,2,5 are articulation points
- More formally:
  - A node a in G is called articulation node if {a} is a separating node-set in a connected component of G
- Obviously, a graph G with at least 3 nodes is biconnected, if and only if it is connected and does not contain an articulation point
- Furthermore, a graph G with at least 3 nodes is biconnected, if and only if G does not contain isolated nodes and every pair of nodes is on a common single cycle



Blocks of a Graph

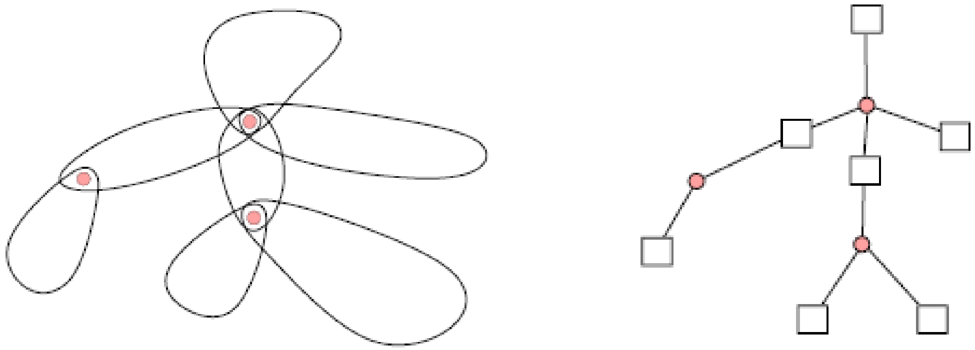
- For two edges $e_1, e_2 \in E$, we define the relation $\equiv$ such that $e_1 \equiv e_2$ if e_1 and e_2 lie on a common simple cycle
- It is easy to see that $\equiv$ defines an equivalence relation on the set of edges, that is E is partitioned into sets E_1, E_2, E_h for a suitable h such that for $e, f \in E_i$ we always have $e \equiv f$
- Let $G_i = (V_i, E_i)$ be the subgraphs induced by E_i for all i = 1, …, h
- These subgraphs are called **blocks**, and blocks that contain at least 2 edges are the maximum sized biconnected components of G
- The graph on the right consists of 3 biconnected components of maximum size with different line styles visualizing the different components



Block Structure of a Graph

- **Lemma 1:** Let $G_i = (V_i, E_i)$ be the blocks of G, then we have
  - For all $i \neq j$: $| V_i \cap V_j | \leq 1$
  - A node $a \in V$ is an articulation node if and only if $V_i \cap V_j = \{a\}$ for suitable $i \neq j$
- The graph on the preceding slide has two articulation points that are each member of exactly two biconnected components
  - Note, that an articulation node can be a member of more than two biconnected components (see next example)

Block Structure Graph B(G) of an Undirected Graph G



- We can describe the block structure of a graph with a so-called **block structure graph** B(G) that contains nodes v_a for each articulation node a and nodes v_b for each block b with each v_a being connected to the nodes v_b denoting the respective biconnected components b that node a is connected to in the original graph
- **Lemma 2:** G is an undirected graph $\Rightarrow$ B(G) does not contain any cycle
  - Proof: If B(G) contained a cycle, we could find a simple cycle in G that contains nodes from different blocks

Computing the Block Structure of a Graph

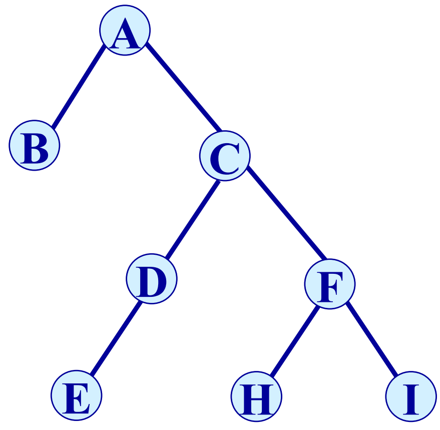
- Why are we interested in articulation nodes and blocks of a graph?
  - If we could identify articulation nodes then we could also compute the blocks of a graph
  - Furthermore, in networking applications we would know which nodes are more important to protect, or between which components of the network we need additional links (however, we would not yet know which links might be wise choices)
- Articulation vertices can be found in $O(|V| (|V| + |E|))$:
  - Just delete each node and do a **depth first search DFS** (see below) on the remaining graph to see if it is (still) connected
- We will see that the articulation nodes and blocks of a graph can also be computed more efficiently by a modified DFS, first proposed by Tarjan in 1972 [Tar72]:
  - For reasons of simplicity, we assume that the graph is connected
  - The blocks of unconnected graphs can be computed by performing the algorithm for each connected component

Depth First Search

- We start with an arbitrary node s and mark it as visited (all other nodes have been marked unvisited before)
  - If (and as long as) the currently considered node v has an unvisited neighbor w:
    - The edge (v, w) is called a tree edge, and v is called parent node of w in the DFS tree
    - We continue our search at w, that is we mark w as visited and it becomes the new current node
  - If (when) the currently considered node v has no (more) unvisited neighbors, we are done in this part of the graph and the parent node of v becomes the new current node
- Runtime: As all nodes and all edges have to be considered exactly once, each time requiring a constant amount of effort, the running time is $O(|V| + |E|)$.
- Memory: In the worst case, the procedure call stack has a depth of $|V|$

Depth First Search

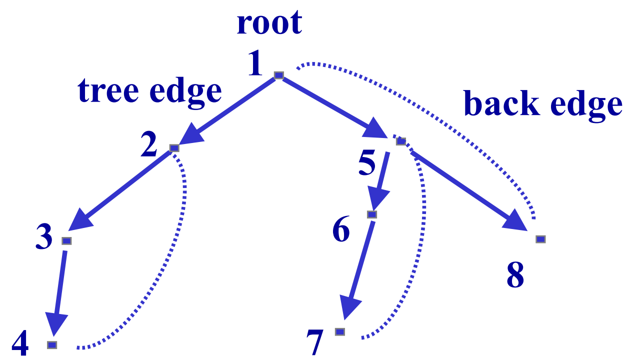
```
int pre = 1, post = 1;
DFS(Node node) // visits all nodes & compute T
{   Node v;
    node.SetPre(pre++);
    for (v = node.First(); v = v.Next(); v != NULL) {
        if (v.GetPre() == -1)
          { node.addTreeChild(v); DFS(v); } }
    node.SetPost(post++);
}
```



- For each node GetPre() & GetPost() return the value of the **preorder** and **postorder number**, respectively, in the example above:
  - Preorder: A, B, C, D, E, F, H, I
  - Postorder: B, E, D, H, I, F, C, A

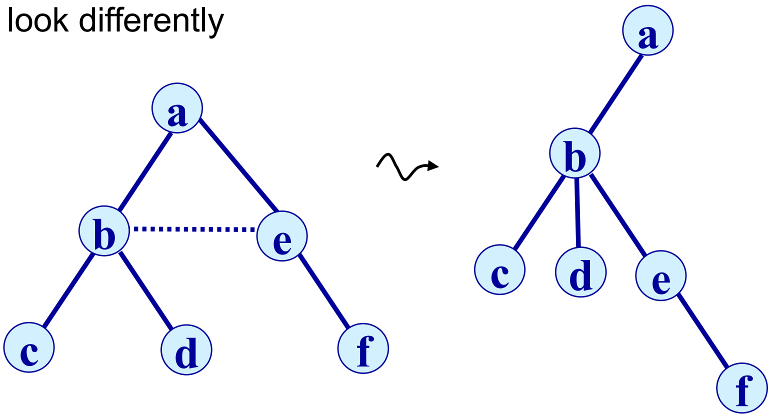
For Connected Graphs DFS Computes a Spanning Tree

- If G is connected, the algorithm DFS(s) computes a spanning tree T of G starting at the start node s (we consider directed edges in the tree T)
- Classification of edges of G with respect to a spanning tree:
  - An edge (v, w) of T is called a **tree edge**
  - An edge (v, w) of G \ T is called a **back edge** if v is a descendent or ancestor of w
  - Else (v, w) is called a a **cross edge** (can not exist in DFS-computed T)

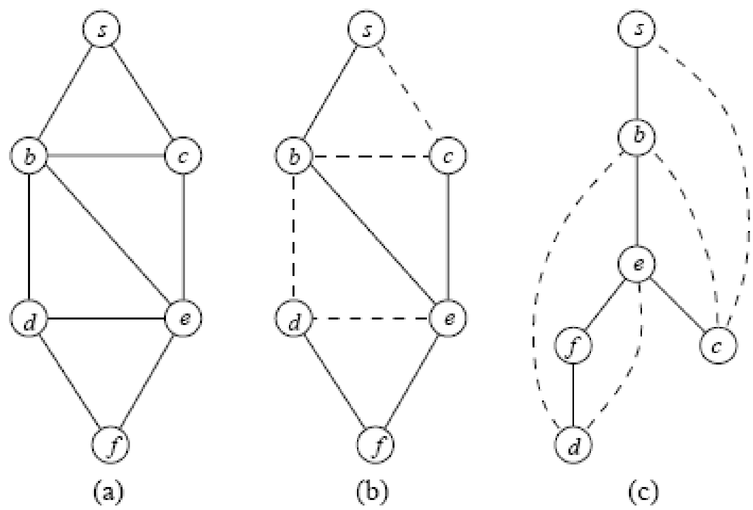


No Cross Edges in DFS Trees for Undirected Graphs

- **Corollary 1:** If DFS is run on an undirected graph, the resulting tree is of a form that there are no edges in G \ T to be classified as cross edges
- Proof idea:
  - Assume that we obtain a tree, so that the edge (b, e) in G \ T is to be classified as a cross edge
  - In this case, we should have considered (b, e) when dealing with b
  - Thus, the tree has to look differently



Resulting Tree, Tree Edges & Back Edges for a Graph G



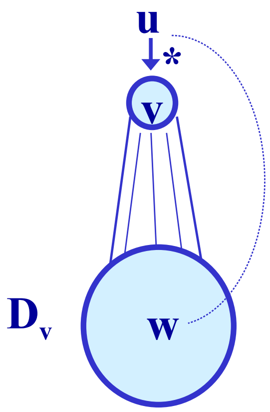
(a) Original graph (DFS computed from node s)

(b) Classifying edges into tree edges (solid) and back edges (dashed)

(c) Resulting DFS tree

Verifying Descendant Relationships via Preordering

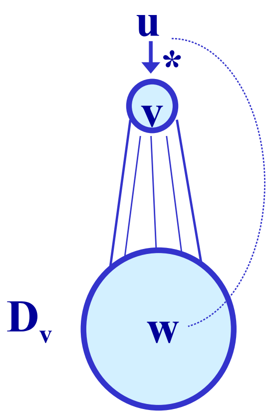
- If there is a directed path from a node u to a node v in T:
  - u is called an ancestor of v
  - v is called a descendant of u
- Suppose we preorder-number a tree T
  Let D_v = # of descendants of v
- **Lemma 3:**
  w is descendant of v
  $\Leftrightarrow$ v.pre < w.pre $\leq$ v.pre + D_v
- In the following, we will further consider single back edges from a descendant w of v to potential ancestors u of v

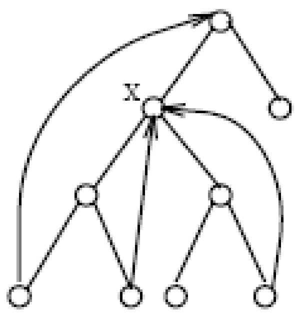


$u \rightarrow v$	:$\Leftrightarrow$	(u, v) is tree edge of T
$u \xrightarrow{*} v$	:$\Leftrightarrow$	u is an ancestor of v
w --- u	:$\Leftrightarrow$	(w, u) is back edge if (w, u) $\in$ G \ T with either $w \xrightarrow{*} u$ or $u \xrightarrow{*} w$

Testing for Proper Ancestors via Preordering

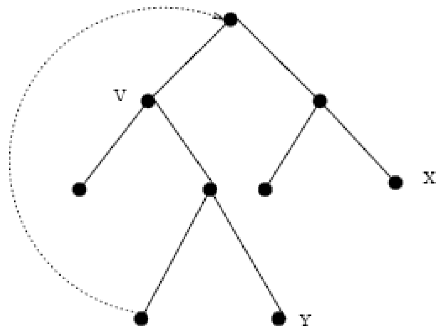
- **Lemma 4:**
  If w is descendant of v and
  (w, u) is back edge such that u.pre < v.pre
  $\Rightarrow$ u is a proper ancestor of v
- **Theorem 1:**
  In a DFS tree T, a node v other than the root is an articulation node if and only if
  - v is not a leaf, and
  - some subtree of v has no back edge incident to a proper ancestor of v
- In the example, x is an articulation node, as its right subtree does not have a back edge incident to a proper ancestor of x
- We have to deal specifically with the root as it does not have any ancestors (later)





Articulation Nodes and Back Edges to Ancestors

- Proof:
  - ⇒ If v is an articulation node then v cannot be a leaf in T. Why?
    - Deleting v must separate a pair of vertices x and y. Because of the other tree edges, this cannot happen unless y is a decendant of v.
  - Let v be a non-root articulation point that separates the nodes x and y
    - Thus, there can not exist a back edge from the subtree that contains y to a proper ancestor of v as this would enable an alternative path between x and y
  - ⇐ If conditions are met for a node v, then it separates any ancestor of v from any descendant in the appropriate subtree, thus v has to be an articulation node



- In order to explain how to efficiently implement this test, we first define the so-called **low value** for every node v

TELEMATIK Rechnernetze

Low Value of Nodes

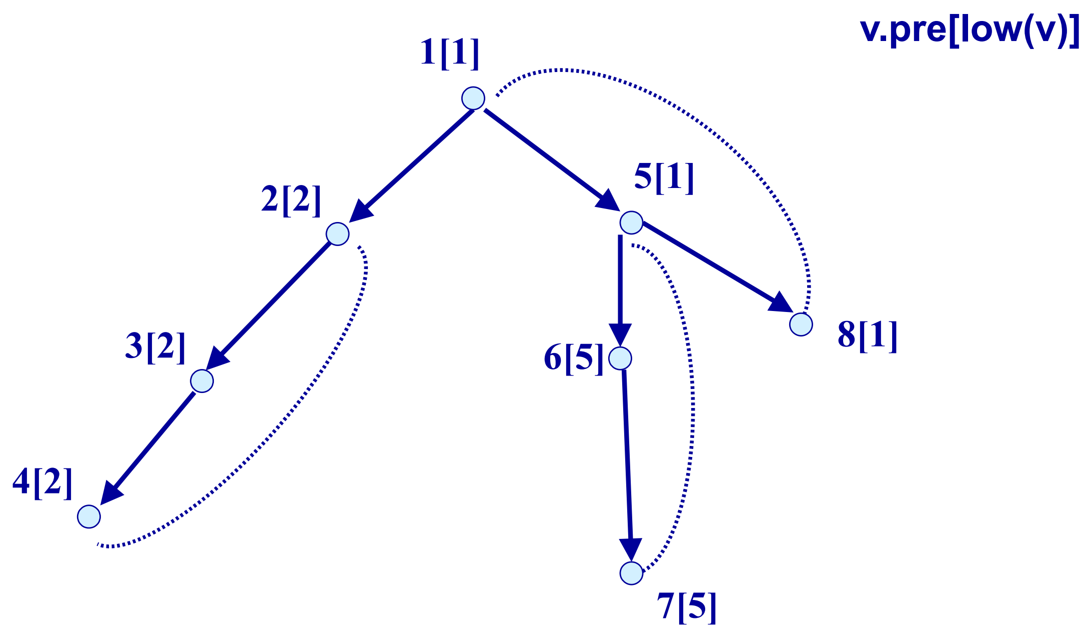
- **Definition:**

  For each vertex v, we define low(v) = min ({v.pre} ∪ {w.pre | $v \xrightarrow{*}$ --- w})
  - By $v \xrightarrow{*}$ --- w we mean that v is connected to w through a path of tree edges and potentially one additional back edge as the last edge
- **Lemma 5:**

  low(v) = min ({v.pre} ∪ {low(w) | v→ w} ∪ {w.pre | v--- w})

- Note:
  - While in the original definition of low(v), the second set considers tree paths of arbitrary length, the terms in the lemma only consider tree paths of length one, that is direct descendants of v (plus potentially one additional back edge)
  - Also, in the second set of the term in the lemma, it is assumed that low(w) has been properly computed for all direct descendants w of v
    (this calls for a nice induction proof that actually delivers the algorithm idea)

Example Graph with DFS Preorder and Low Values



Computing Low Values of Nodes

- Lemma 5 enables us to compute low(v) for all nodes v by using DFS and evaluating preorder values of incident nodes as we visit each node
- For each node v that is visited during a DFS we set v.pre, initialize v.low := pre.v and consider all edges of v:
  - For tree edges to unvisited nodes w, we perform a recursive call and after it returns and w.low has been computed properly, we set v.low := min (v.low, w.low)
  - For back edges to nodes w that have already been visited, we set v.low := min (v.low, w.pre)
- The following theorem tells us how to use the values low.v and pre.v in order to determine the articulation nodes of G

Computing Articulation Nodes (1)

- **Theorem 2:**
  A node a is an articulation node if and only if either
  - The node a is the DFS tree root with ≥ 2 tree children, or
  - The node a is not the DFS tree root but it has a tree child v with low (v) ≥ a.pre
- Proof:

⇐
  - Assume that node a is the DFS tree root with at least two tree children. As in a DFS tree there are no cross edges (corollary 1), all paths between nodes in the subtrees originating at node a have to go over a. Thus, node a is an articulation node.
  - Assume that node a is not the root of the DFS tree but it has a tree child v with low(v) ≥ a.pre. This implies that there is no back edge from nodes below node a to a proper ancestor of node a. By theorem 1 we know that node a has to be an articulation point.

Computing Articulation Nodes (2)

- Proof (continued):

⇒
  - If node a is the DFS tree root and is an articulation node, it must have ≥ 2 tree edges to two distinct biconnected components. Otherwise the graph would remain connected after removal of node a (and node a was no articulation node).
  - If node a is not the DFS tree root and is an articulation node, node a must have a child node b, such that the subtree originating in node b contains all nodes of a connected component of graph G \ {a}. For this node b, it must hold that b.low ≥ a.pre
- Thus, we can compute the articulation nodes of a graph with a slightly modified DFS in time O(|V| + |E|)

Computing the Blocks of a Graph

- In order to also compute the blocks of graph G while computing its articulation points we introduce an additional stack s of edges:
  - Whenever we find a tree edge (v, w), we push it to the stack s prior to making the recursive call DFS(w)
  - Whenever we find a back edge, we also push it to the stack s
  - Whenever a recursive call for node w returns to the parent node v and we have w.low ≥ v.pre, then all edges on top of the stack up to the edge (v, w) form the next identified block
- As for each node and for each edge we only have to perform a constant number of steps the overall running time of the algorithm is O(|V| + |E|)

bicon(s, s) computes set A of articulation nodes and set B of blocks

```
procedure bicon(Node v; Node pv) {  // v current node, pv parent of v
v.visited = true;
v.pre = v.low = current++;
int c = 0; // counts number of children of node v in DFS tree
foreach (neighbor edge e = (v, w) of v) {
  if (w.visited == false) { // tree edge
     s.push(e); c++; bicon(w, v); // recursive call
     v.low = min(v.low, w.low);
     if ((w.low ≥ v.pre) and ((v != s) or (c = 2))) { A.insert(v); } // v is articulation node
     if (w.low ≥ v.pre) {
        Edge f; Set C := ∅;
        for (f = s.pop(); f != e; f = s.pop()) {C.insert(f);}
        C.insert(f);
        B.insert(C); } // of tree edge
  else if ((w.pre < v.pre) and (w != pv)) { // back edge
            S.push(e); v.low = min(v.low, w.pre); }
  } // of for all neighbor edges
} // of bicon(Node v, Node pv)
```

```
Algorithmus zur Berechnung der Blöcke eines Graphen
Eingabe: zusammenhängender, ungerichteter Graph G = (V, E)
Ausgabe: Menge A der Artikulationsknoten von G, Menge B der Blöcke von G

procedure bicon(node v, node pv)
begin // v ist aktueller Knoten, pv sein Vater
    visited[v] := true;
    pre[v] := low[v] := current;
    current++;
    int c := 0; // zählt Anzahl Kinder von v im DFS-Baum
    for alle Nachbarkanten e = {v, w} von v do
        if visited[w] = false then // Baumkante
            S.push(e);
            c++;
            bicon(w, v); // rekursiver Aufruf
            low[v] = min(low[v], low[w]);
            if low[w] ≥ pre[v] and (v ≠ s or c = 2) then
                A := A ∪ {v}; // v ist Artikulationsknoten
            fi
            if low[w] ≥ pre[v] then
                C := ∅;
                repeat
                    f := S.pop();
                    C := C ∪ {f};
                until f = e;
                B := B ∪ {C}; // C ist der nächste Block
            fi;
        else if pre[w] < pre[v] and w ≠ pv then // Rückwärtskante
            S.push(e);
            low[v] = min(low[v], pre[w]);
        fi;
    od;
end;

// Hauptprogramm
for alle v ∈ V do visited[v] := false; od;
s := ein beliebiger Startknoten;
current := 1; A := ∅; B := ∅;
S := leerer Stack;
bicon(s, s);
return (A, B);
```

Algorithm Pseudocode from [ERL02]

Additional References

[Erl02]	Thomas Erlebach. *Algorithmen für Kommunikationsnetze.* Lecture Script, ETH Zürich, 2002.
[Tar72]	Robert Endre Tarjan. *Depth-First Search and Linear Graph Algorithms.* SIAM Journal of Computing, 1, pp. 146-160, 1972.
[ET76]	Kapali P. Eswaran and Robert Endre Tarjan. *Augmentation Problems.* SIAM Journal of Computing, 5(4), pp. 653-665, 1976.
[NI92]	Hiroshi Nagamochi and Toshihide Ibaraki. *Computing Edge-Connectivity in Multigraphs and Capacitated Graphs.* SIAM Journal of Discrete Mathematics, 5(1), pp. 54-66, February 1992.
[SW97]	Mechthild Stoer and Frank Wagner. A Simple Min-Cut Algorithm. Journal of the ACM, 44(4), pp. 585- 591, July 1997.