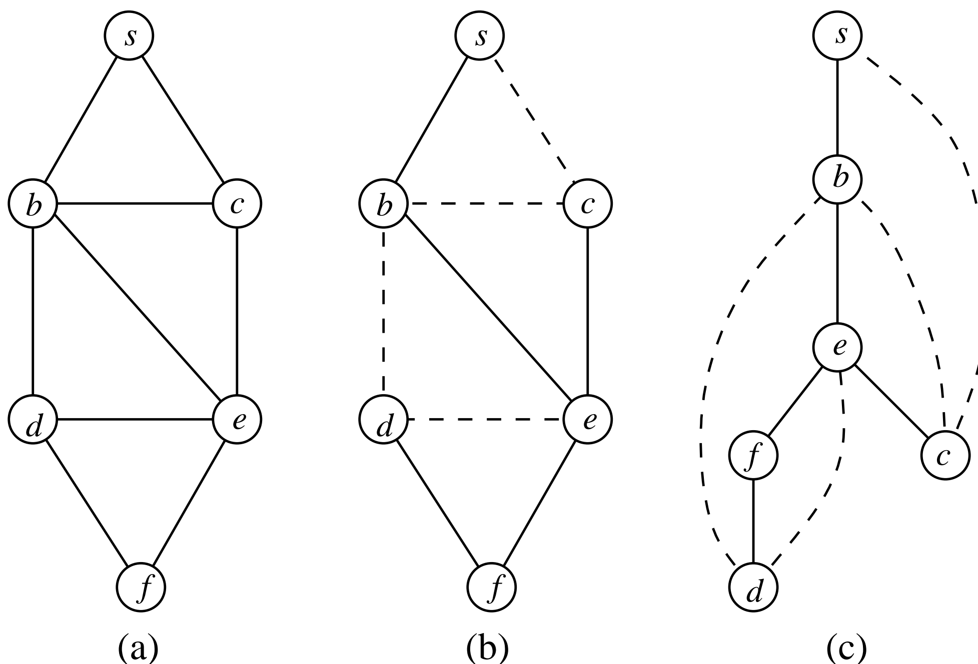
1 DFS-trees in undirected graphs

Let $G = (V, E)$ be an undirected connected graph, and let $s \in V$ be some node of G. If we start a depth first search (DFS) at s, then this DFS visits all nodes of G. Each edge $e \in E$ can exhibit one of two possible characteristics:

1. The DFS visits a so far unvisited node w using the edge $e = \{v, w\}$; in this case, we call e a *tree edge* which is considered being directed (downwards) from v to w.
2. The DFS considers the edge e but the node on the other side of the edge is already visited; in this case, we call e a *back edge*.

We illustrate this with an example:



The picture shows (a) an undirected graph, (b) the partition of the edges into tree edges (solid lines) and back edges (dashed lines), and (c) the rooted variation of the obtained DFS-tree.

2 Biconnected components

Let $G = (V, E)$ in the following be an undirected connected graph.

Definition 1 (Connectivity). *The (vertex) connectivity of G is the size of the smallest subset of nodes such that after removing these nodes the resulting graph is unconnected.*

If G is a clique of size n then its connectivity is defined as $n - 1$.

A graph having connectivity of at least 2 is called *biconnected*.

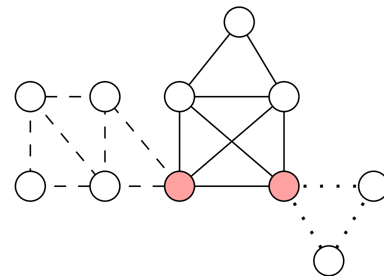
Definition 2 (Articulation vertices)**.** *A node a of G is called* articulation vertex *if the number of connected components of G increases by removing a from G.*

It is easy to see that G is biconnected if and only if G has at least three nodes, is connected and doesn't have an articulation vertex.

Definition 3. *The* biconnected components *of a graph are the maximal biconnected induced subgraphs.*

A block is a maximal connected induced subgraph without any articulation node (w.r.t. the subgraph itself).

That means, the set of all blocks consists of all biconnected components, all bridges and all isolated nodes.



The accompanying graph consists of three blocks which are denoted by solid, dashed, and dotted lines, respectively. The articulation vertices are shaded.
By the way: The intersection of two blocks contains at most one single node. Therefore, no edge can belong to two different blocks. The nodes which are contained in the intersection of two different blocks are exactly the articulation vertices. The blocks constitute a tree structure, that means, the structure we obtain by considering blocks that share an articulation vertex as neighbours is cycle-free.

The connectivity of a graph and the computation of blocks is (for example) relevant when building a failure-resistant communication network. A biconnected subnetwork will continue to be connected even if one of its nodes is removed.

2.1 Modified DFS-algorithm

The algorithm presented in this section can find all blocks in a connected graph G. If we are only interested in the biconnected components we just have to filter out all bridges. If the graph isn't connected, we apply the algorithm individually to each connected component.

By applying a modified depth first search to G we can compute the following values:

- $pre[v]$ = preorder number (DFS-number) of v, i.e., of all nodes visited by the DFS, what number v was ($pre[s] = 0$).
- $low[v]$ = minimal preorder number $pre[w]$ of a node w which can be reached by starting at v, then following ≥ 0 tree edges, and then *possibly* using one single back edge.

The calculation of the *low*-values can be done by utilizing the following observation:

$$low[v] = \min\Big(\{pre[v]\} \cup \{low[w] \mid w \text{ is a child of } v \text{ in the DFS-tree } \} \cup \{pre[w] \mid \{v, w\} \text{ is a back edge } \} \Big)$$

More precisely, at the beginning of the recursive DFS-function, we calculate for a node v the value $pre[v]$ and initialize $low[v]$ by $pre[v]$; then we consider all indicent edges of v: if an edge is a tree edge from v to w, then we induce a recursive call of the function on and set $low[v] = \min\{low[v], low[w]\}$ after that; if the edge is a back edge from v to w then we set $low[v] = \min\{low[v], pre[w]\}$.

We can now apply the following Lemma.

Lemma 4. *Let $G = (V, E)$ be an undirected connected graph and T a DFS-tree in G. A node $a \in V$ is an articulation vertex if and only if either*
(a) a is the root of T and has at least two children, or
(b) a is not the root of T and a has a child b with $low[b] \geq pre[a]$.

Therefore, the articulation vertices of G can efficiently be calculated in time $O(|V|+|E|)$ by using a slightly modified DFS. The blocks can also be computed in linear time: We use a stack S of edges which is empty at the beginning. If a tree edge is found, then we put it on top of the stack S before we induce the recursive call of the function; if a back edge is found, then we immediately put it on top of S. Whenever we backtrack from the recursive call on node w to the parent node v and $low[w] \geq pre[v]$ holds, then all the edges on top of the stack until (including) edge $\{v, w\}$ constitute exactly the edges of a block.

3 DFS-forests in directed edges

Let $G = (V, E)$ be a directed graph. If we start a DFS at a node of G, then we don't necessarily reach all nodes, and we have to start another DFS at some unvisited node. Therefore, a DFS of a directed graph does in general not yield a single tree but a forest consisting of several trees. Each directed edge $e = (v, w) \in E$ is considered by the DFS if the node v is processed, and can exhibit one of four possible characteristics:

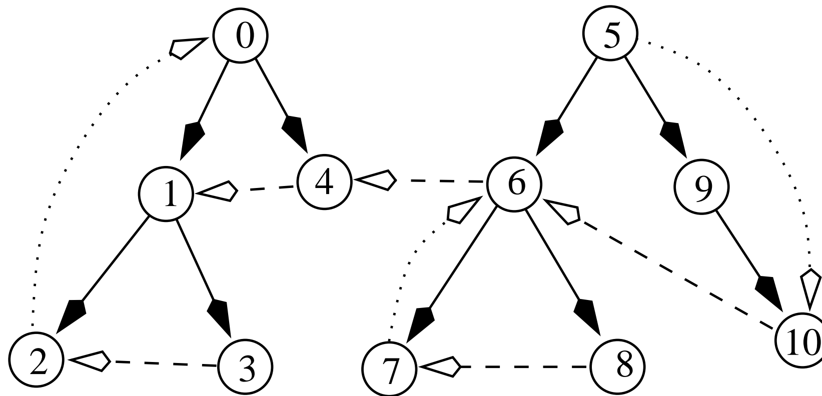
1. w is unvisited; in this case, we call e a *tree edge* which is considered being directed (downwards) from v to w.

2. w is already visited and its DFS-number is larger than the DFS-number of v; in this case, we call e a *forward edge.*

3. w is already visited and is an ancestor (parent, grandparent, etc.) of v with respect to the same DFS-tree; in this case, we call e a *back edge.*

4. w is already visited and is not an ancestor (parent, grandparent, etc.) of v and also not a descendant (child, grandchild, etc.) of v with respect of the same DFS-tree; in this case, w can be part of the same DFS-tree as v or can be part of a DFS-tree which was created before, and we call e a *cross edge*.

We illustrate the different kinds of edges in the following example:



The picture shows a possible DFS-forest which can be obtained by a DFS in a directed graph. Tree edges are solid, cross edges are dashed, forward and back edges are dotted. Each node contains its respective DFS-number (preorder number).

4 Strongly connected components

Let $G = (V, E)$ be a directed graph. Two nodes x and y of G are part of the same *strongly connected component* if there is a directed path from x to y and a directed path from y to x in G. Using this implicit definition we can partition the set V of nodes of G into strongly connected components. The example graph above consists of the five strongly connected components $\{0, 1, 2, 3, 4\}$, $\{6, 7, 8\}$, $\{10\}$, $\{9\}$ and $\{5\}$. For a given graph $G = (V, E)$, we want to compute all strongly connected components in time $O(|V| + |E|)$. To this end, we can again use a modified DFS.

We consider the DFS-forest and the preorder numbers obtained by a DFS in G. For each strongly connected component $Z \subseteq V$ we call the node of Z having the smallest preorder number *root* the strongly connected component. We can show that a node r is a root if and only if it satisfies the following properties:

1. there is no back edge from r or from a descendant of r (child, grandchild, etc.) to an ancestor (parent, grandparent, ect.) in the DFS-forest

2. there is no cross edge from r or a descendant of r to a node w such that the root of the strongly connected component of w is a ancestor of r

(For instance, the node 4 of the example graph above has such a cross edge and therefore is not the root of a strongly connected component.) The strongly connected components can now be "picked" from the DFS-forest by iteratively and in a bottom-up fashion

removing a root from the forest together with its subtree which is attached to the root (in the example above the subtrees being attached to the roots 0, 6, 10, 9, and 5, in this order). This can be done by a single DFS (in the sense that each node and edge has to be considered only a constant number of times).

In addition to the usual preorder number $pre[v]$ of a node v we define a value $lowlink[v]$:

- $lowlink[v]$ = minimal preorder number $pre[w]$ of a node w that can be reached by starting at v, using ≥ 0 tree edges and possibly one back edge after that, or that can be reached by starting at v, using ≥ 0 tree edges followed by a single cross edge such that the root of w's strongly connected component is an ancestor of v.

The *lowlink*-valus can be computed during a slightly modified DFS using the following equality:

$$
\begin{aligned}
lowlink[v] \ = \min \Big(\ & \{ \ pre[v] \ \} \cup \{ \ lowlink[w] \mid w \text{ is child of } v \text{ in the DFS-forest } \} \\
& \cup \ \{ \ pre[w] \mid (v,w) \text{ is a back edge } \} \\
& \cup \ \{ \ pre[w] \mid (v,w) \text{ is a cross edge, and the root of the} \\
& \qquad \text{strongly connected component of } w \text{ is an ancestor of } v \ \} \Big)
\end{aligned}
$$

More precisely, at the beginning of the recursively defined DFS-function we calculate the value $pre[v]$ of the current node v and initialize $lowlink[v]$ by the value of $pre[v]$; then we consider all edges emanating from v: if an edge is a tree edge to a child w, then we make a recursive call of the DFS-function on w and set $lowlink[v] = \min\{lowlink[v], lowlink[w]\}$ afterwards. If an edge is a back edge then we set $lowlink[v] = \min\{lowlink[v], pre[w]\}$. If an edge is a cross edge to a node w, then we must decide if the root of the strongly connected component of w is an ancestor of v; this is the case if and only if w wasn't already assigned to a strongly connected component ("picked"). If this is the case, then we again set $lowlink[v] = \min\{lowlink[v], pre[w]\}$ (otherwise, the cross edge is ignored).

We use the fact that a node r is the root of a strongly connected component if and only if after the execution of the DFS-function for r the condition $lowlink[r] = pre[r]$ holds. Therefore, it is easy for the algorithm to recognize the roots of the strongly connected components. The strongly connected components itselves can be calculated on the fly by using the following approach. We use a stack S of nodes which is empty at the beginning. When the recursive DFS-function is called on a node v, then we put v on top of the stack S at the beginning of the execution of the function. If at the end of the execution of the function for node v the condition $lowlink[v] = pre[v]$ holds, then all nodes on top of the stack until (including) v constitute the next strongly connected component Z. All nodes of Z must be marked as "already assigned to a strongly connected component" such that cross edges that lead to these nodes and are encountered later on are ignored when calculating *lowlink*-values.
Remark: It should be noted that people were able to find algorithms which compute biconnected components in undirected graphs, and strongly connected components in

directed graphs in time $O(|V| + |E|)$, respectively, and which are basically only slightly modified depth first searches. At a first glance we could have assumed that only algorithms with a much larger running time could solve these problems.