Now let T be a tree, and let v be an arbitrary vertex. Apply the proof of Lemma 2.1 to T, setting $r = v$. The spanning tree that we obtain has $n - 1$ edges, and therefore must be again T. For any vertex $w \neq v$, we have a unique path from v to w in $G_T = T$. This proves that for any tree T and any two vertices v, w, there exists a unique path between v and w in T.

Trees have another useful property:

Lemma 2.3 *Any tree with $n \geq 2$ vertices has at least two* leaves, *i.e., vertices with degree 1.*

Proof: Let T be a tree with $n \geq 2$ vertices, and let w be a vertex of minimum degree in T. Because T is connected and $n \geq 2$, vertex w must have at least one neighbor, so $\deg(w) \geq 1$. Assume now that all vertex $\neq w$ have degree ≥ 2, then

$$2m = \sum_{v \in V} \deg(v) \geq 2(n-1) + 1.$$

But on the other hand, $2m = 2(n-1)$ because T is a tree, so this implies $0 \geq 1$, which is impossible. Therefore there must be at least one vertex $x \neq w$ with $\deg(x) \leq 1$. As above, one argues $\deg(x) \geq 1$, so $\deg(x) = 1$. Since w was the vertex of minimum degree, this means $\deg(w) = 1$ as well, which proves the claim. □

2.2 Higher connectivity

2.2.1 Definitions

A *cutting k-set* is a set V_k of k vertices such that $G - V_k$ has strictly more connected components than G. When $k = 1$, the vertex in V_k is known as a *cut-vertex* (sometimes also called *articulation point*), and when $k = 2$, the pair of vertices in V_k is known as a *cutting pair* (sometimes also called *separation pair*). A graph is *k-connected* if there is no cutting set with fewer than k vertices. In Figure 2.1 the leftmost graph is 1-connected, since the circled vertex is a cut-vertex. The middle graph is 2-connected (also known as *biconnected*), while the rightmost graph is 3-connected (also known as *triconnected*). There are no fancy names such as, say, penta-connected for graphs that are k-connected, for $k > 3$.

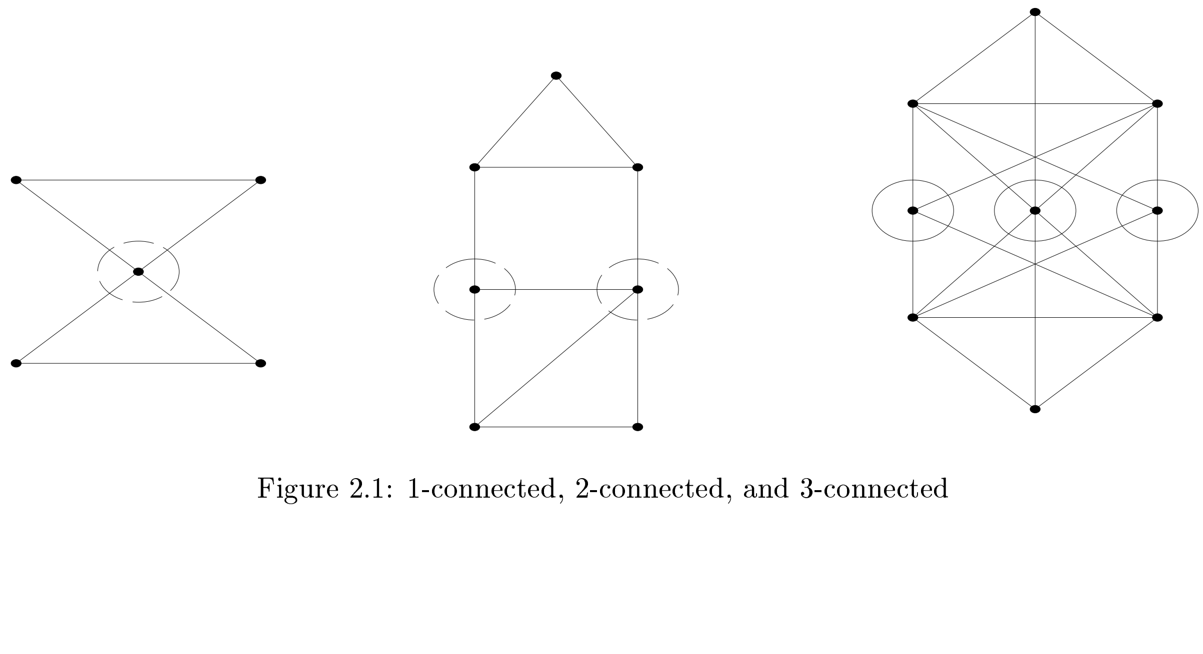

Figure 2.1: 1-connected, 2-connected, and 3-connected

If G is k-connected and $n \geq k+1$, then any vertex must have degree $\geq k$. For if $\deg(v) = l \leq k-1$, then the $\leq l$ neighbors of v separate v from the rest of the graph, which is non-empty by $n \geq k+1$. But even more can be said:

Theorem 2.4 (Menger's theorem) *A graph G with $n \geq k+1$ vertices is k-connected if and only if for any 2 distinct vertices v, w in G, there exist k paths $P_1, \ldots, P_k$ in the underlying simple graph of G that are* internally vertex-disjoint, *i.e., that have no vertex other than v and w in common.*

For a proof of Menger'stheorem, refer for example to [Gib85]. In particular, we can apply Menger's theorem for $k = 2$; then the two paths form a simple cycle.

Corollary 2.5 *A graph G with $n \geq 3$ vertices is biconnected if and only if for any two vertices v, w there exists a simple cycle in the underlying simple graph of G containing both v and w.*

A *biconnected component* is a maximal connected subgraph of G that is biconnected. The graph in Figure 2.2 has four biconnected components, even though the entire graph is connected. We study in Section 2.2.2 how to find biconnected components efficiently.

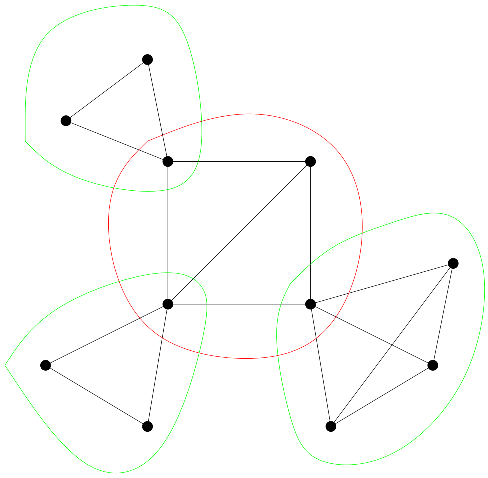

Figure 2.2: A graph with four biconnected components.

Menger's theorem also helps in characterizing when two vertices belong to the same biconnected component.

Lemma 2.6 *For any graph G with $n \geq 3$ vertices, two vertices v and w belong to the same biconnected component if and only if (v, w) is an edge and/or there is a simple cycle in the underlying simple graph of G containing both v and w.*

Proof: If v and w belong to the same biconnected component, then they belong to a biconnected subgraph of G, which either has two vertices (then it is an edge (v, w)), or there exists such a cycle by Corollary 2.5. Conversely, assume that there exists a simple cycle C containing v and w in the underlying simple graph of G. The vertices of C then induce a biconnected subgraph G' of G, because cycle C serves for any pair of them. If (v, w) is an edge, then this edge is a biconnected subgraph G' of G. Either way, by extending G' to a maximal biconnected subgraph of G, we obtain one biconnected component, so v and w belong to the same biconnected component. □

Using this characterization, we can analyze the structure of biconnected components.

Lemma 2.7 *Any vertex that is not a cut-vertex belongs to exactly one biconnected component. On the other hand, any cut-vertex belongs to at least two biconnected components.*

Proof: Assume that the graph is connected, otherwise we can show the claim for each connected component separately. Let v be a vertex that is not a cut-vertex, and let v_0 and v_1 be two neighbors of v. Since $G - v$ is connected, there exists a path from v_0 to v_1 in $G - v$, which can be completed with the edges to v to a simple cycle containing (v, v_0) and (v, v_1). Hence any two incident edges of v belong to the same biconnected component.

For the other claim, assume that v is a cut-vertex, and let $G_1, \ldots, G_k$, $k \geq 2$, be the connected components of $G - v$. Let $x_1 \in G_1$ and $x_2 \in G_2$ be two neighbors of v in G_1 and G_2, respectively; these exist because G is connected. If there were a simple cycle in G containing (v, x_1) and (v, x_2), then this would give at least a path in $G - v$ connecting x_1 and x_2, contradicting that they are in different connected components of $G - v$. So there are two incident edges of v that belong to two different biconnected components. □

So two biconnected components can intersect only in a cut-vertex. This motivates the following definition: Let the *block-graph* G_B be the graph that has a vertex v_B for every biconnected component B of G and a vertex v_w for every cut-vertex w of G. Connect v_w and v_B if and only if w is a cut-vertex that belongs to B. It can be shown that G_B is a tree; therefore G_B is usually called the *block-tree*.

In particular this implies that no two biconnected component have more than one cut-vertex in common. Since any vertex that is not a cut-vertex belongs to only one biconnected component, this means that no two biconnected components have more than one vertex in common, which implies that every edge belongs to only one biconnected component.

This ends the discussion of biconnected components.

There is such a thing as a *triconnected component*, but its definition is *not* a maximal subgraph that is triconnected, but something a little more complicated. As we will not need this, we skip the definition, and refer instead to [HT73] and [DBT96]. There, it is also shown how to compute these triconnected components efficiently (in $O(n + m)$ time), and how to store them for effective processing in a data structure called an *SPQR-tree*.

Edge-connectivity

The previous definitions were concerned with *vertex-connectivity*, which is characterized by how many vertices we can take away without splitting the graph. There is the related concept of *edge-connectivity*, which is characterized by how many edges we can take away without splitting the graph. We will only need one definition concerning edge-connectivity:

Definition 2.8 *An edge e is called a* bridge *of graph G if $G - e$ has more connected components than G. A graph is called* bridgeless *if it has no bridge.*

2.2.2 Computing biconnected components

Now we show how to find biconnected components in a simple graph. We assume that the graph is connected; if it is not then we can compute the connected components as explained earlier, and run the algorithm on each connected components. We also assume that $n \geq 2$; any graph with one vertex is biconnected.

Depth-first search and low-numbers

The algorithm for computing biconnected components is based on a depth-first search (DFS) of the graph. Depth-first search is a well-known graph traversal algorithm, therefore we only mention some of its important properties without proofs and establish notation.

A depth-first search numbers vertices with numbers $1, 2, \ldots, n$ in order in which they were visited for the first time. The number assigned by the depth-first search to a vertex v will be called *DFS-number* and denoted $num(v)$. A depth-first search computes also a spanning tree of the graph called *DFS-tree*. A DFS-tree contains an edge e if the depth-first search has discovered a new vertex following this edge. Edges of the graph contained in the DFS-tree are called *tree-edges*. Other edges of the graph are called *back-edges*.

We will repeatedly use the following property of the DFS-tree: if (v, w) is a back-edge, then either v is an ancestor of w or w is an ancestor of v.

We associate the following orientation with edges: tree-edges are oriented downwards from parent to a child and back-edges are oriented upwards to the ancestor. We will use $v \to w$ to denote a tree-edge, and $v\!- \to w$ to denote a back-edge. Figure 2.3(a) shows a graph and the DFS-numbers of its vertices.

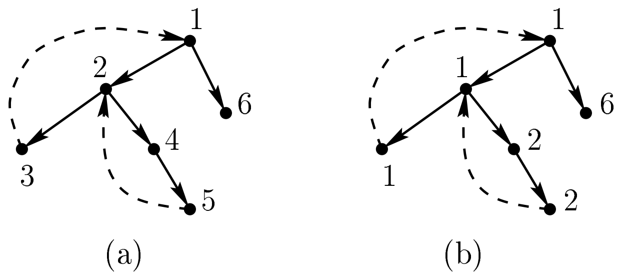

Figure 2.3: Illustration of a DFS-tree. Tree-edges are solid lines and back-edges are dashed lines.

Finally we define low-numbers. Low-numbers are computed in the first pass of the algorithm for finding biconnected components and they are used in both passes.

Definition 2.9 *The* low-number *of a vertex v (denote $low(v)$) is the smallest DFS-number of a vertex that can be reached from vertex v by going down using several (possibly zero) tree-edges and then optionally one back-edge up.*

See Figure 2.3 for an example. Computing the low-numbers directly from this definition would be inefficient. Therefore we give also a recursive characterization of low-numbers in Lemma 2.10 that allows a more efficient implementation.

Lemma 2.10 *Let v be a vertex. Then*

$$\begin{aligned} low(v) = \min\{ \; & num(v), \\ & num(w), \text{ where } v\!- \to w \text{ is a back-edge}, \\ & low(x), \text{ where } v \to x \text{ is a tree-edge}\} \end{aligned} \tag{2.1}$$

Proof: According to the definition of low-numbers $low(v) = num(w)$ for some vertex w. There are several possible cases.

First assume that we can get from v to w without using any tree-edge. This means that either $w = v$ or w is an ancestor of v directly connected to v by a back-edge. These two cases are covered by the first two lines in Formula 2.1.

If this is not the case, then $low(v)$ is the smallest DFS-number of a vertex that can be reached from vertex v by going to one of its children x, then further down using several (possibly zero) tree-edges and then optionally one back-edge up. In this case, $low(v) = low(x)$, which is also covered by Formula 2.1. □

Pass 1 of the algorithm

The first pass of the algorithm computes the DFS-tree, DFS-numbers of the vertices and also the low-numbers. The latter are computed for v once we are finished with computing DFS-numbers and low-numbers for all children of v according to Lemma 2.10. Thus, we compute $low(v)$ when we retreat from v. Using this information, we determine whether v is a cut-vertex with the following two rules:

- If v is the root of the DFS-tree, then v is a cut-vertex if and only if it has at least two children.
- If v is not the root of the DFS-tree, then v is a cut-vertex if and only if it has a child w with $low(w) \geq num(v)$.

All computations for vertex v take $O(\deg(v))$ time, and therefore the entire algorithm works in $O(m)$ time.

Now we will prove the correctness of the criterion used in the algorithm for determining whether given vertex is a cut-vertex.

Lemma 2.11 *Let v be a vertex that is not a root of the DFS-tree. Then vertex v is a cut-vertex if and only if v has at least one child w such that $low(w) \geq num(v)$.*

Proof: Assume that v is a vertex that is not a root of the DFS-tree, and w is a child of v such that $low(w) \geq num(v)$. Since v is not a root of the tree, it has a parent $p(v)$ (see Figure 2.4(a)). We will show that when we remove vertex v from the graph, vertices $p(v)$ and w will not be connected by any path, therefore v is a cut-vertex.

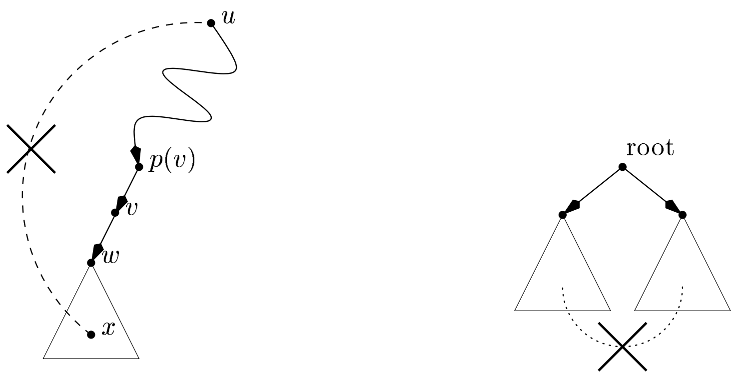

Figure 2.4: Cases in which a vertex is a cut-vertex.

From the properties of the DFS-tree, any edge leading from the subtree rooted in w can have its other end either inside this subtree or in some ancestor of w. However if there were an edge leading from a vertex x in the subtree to some ancestor u of v, it would be possible to go from w down the tree to x and then one back-edge up to the u, hence $low(w) \leq num(u) < low(v)$ contrary to the assumption. that $low(w) \geq num(v)$ because $num(u) < num(v)$. Thus all edges from the subtree rooted in w end at v or in the subtree rooted at w, and if we remove v, this subtree is not connected to $p(v)$.

Now assume that vertex v is a cut-vertex. We will show that it has at least one child w such that $low(w) \leq num(v)$. Since v is a cut-vertex, after deleting it from the graph we will get at least two connected components. Since all ancestors of v belong to one such connected component, at least one child of v that will be in a different component than $p(v)$. And this child is the vertex w that we are looking for. Since deleting v disconnects w from the rest of the tree, there is no edge between the subtree rooted in w and either $p(v)$ or some of its ancestors. Therefore $low(w) \geq num(v)$. □

Lemma 2.12 *The root of the DFS-tree is a cut-vertex if and only if it has at least two children.*

Proof: Imagine that we remove the root of the DFS-tree from the graph. Each of its children will be in a different connected component (see Figure 2.4(b)), because no edge connects subtrees rooted in the children of the root (a back-edge connects a vertex with its ancestor). Therefore if the root has at least two children, it is a cut-vertex.

If the root is a cut-vertex then removing it yields at least two connected components. The depth first search starts in the root and enters one of these components. It cannot get to the other component other than to retreat back to the root and then to continue in the other component. Therefore the root has at least two children. □

If we only want to know whether the graph is biconnected, and if not, which vertices are cut-vertices, then we can stop after the first pass.

Pass 2 of the algorithm

In the second pass of the depth-first search we compute biconnected components of the graph. First we have to decide how to store these components. Recall that each edge belongs to only one biconnected component. A vertex, on the other hand, can belong to several components if it is a cut-vertex. The following data structures can be used for storing biconnected components:

1a. For each biconnected component keep a list of edges belonging to this component.

1b. Label each edge so that edges belonging to the same component have the same label while edges belonging to different components have different labels.

2. Explicitly split the graph into biconnected components. Compute the block-tree, and store each biconnected component B at v_B.

Possibilities 1a and 1b are essentially the same. We will use 1b. which is slightly easier to implement because we do not need to build any new data structures. Possibility 2 can be done in linear time once we have marked the edges with labels.

The algorithm traverses the vertices in DFS-order and keeps in a global variable NOB for the number of biconnected components found so far. When it visits a vertex v, it labels all edges outgoing from v, i.e., we label a tree-edge when inspecting the parent, and a back-edge when inspecting the vertex with the higher DFS-number. Each edge either starts a new biconnected component (then we increase the variable NOB and assign this new value to the edge) or it belongs to the same biconnected component as the edge $(p(v), v)$ (then we label the edge with the same label as $(p(v), v)$); this of course is possible only if v is not the root. This is decided with the following rule: The edge $v \to w$ starts a new biconnected component if and only if it is a tree-edge and $low(w) \geq num(v)$.

The second pass of the algorithm only scans the vertices in DFS-order and assigns a label to each edge in $O(1)$ time. The time-complexity of the algorithm is therefore $O(n+m)$.

We need to prove that our criterion for assigning labels to edges is correct. We state this in the following lemma.

Lemma 2.13 *The edge $v \to w$ starts a new biconnected component if and only if it is a tree edge and $low(w) \geq num(v)$. Otherwise the edge $v \to w$ is in the same biconnected component as the edge $p(v) \to v$.*

Proof: First, assume that the edge $v \to w$ is a back-edge (see Figure 2.5(a)). In this case w is an ancestor of v (and thus v is not a root). Using tree-edges, we can find a simple cycle $C = w \to \cdots \to p(v) \to v - \to w$. This shows that $v, p(v)$ and w all belong to the same biconnected component (Lemma 2.6). See Figure 2.5(a).

Next, consider the case when the edge $v \to w$ is a tree edge and $low(w) < num(v)$. This means that from the subtree rooted in w there is a back-edge e starting at some descendant x of w, and ending at some ancestor u of v. Using tree-edges, we can find a simple cycle $C = w \to \cdots \to x - \to u \to \cdots \to p(v) \to v - \to w$. This shows that $v, p(v)$ and w all belong to the same biconnected component (Lemma 2.6). See Figure 2.5(b).

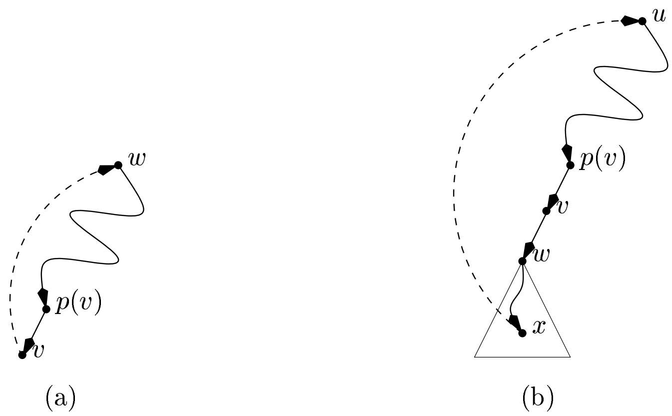

Figure 2.5: Cases where an edge (v, w) does not start a new biconnected component.

Next assume that v is the root of the DFS-tree and w is its child. Then $low(w) \geq num(v)$. If we remove the root, each of its children will be in a different connected component (see the proof of Lemma 2.12). Therefore each child of the root starts a new biconnected components.

In the last case the edge $v \to w$ is a tree edge, v is not a root and $low(w) \geq num(v)$. As we discussed in the proof of Lemma 2.11, this means that after removing vertex v the subtree rooted in w will not be connected to the rest of the graph. It means that all edges belonging to the same component as $v \to w$ are inside the subtree rooted in w. We process vertices in DFS-order and therefore none of these edges were labeled so far. We can therefore start a new biconnected component. $\square$

One-pass algorithm

We have presented the algorithm which finds biconnected components in two passes. It is possible, but not trivial, to do the computation in one pass. The problem is that in our algorithm we need to know the label of the edge $p(v) \to v$ when we process the edges starting in v. However at that time the computation for vertex $p(v)$ is not finished yet and this vertex does not have a label assigned. The solution is to store the edges in a stack and as soon as we finish traversal of one biconnected component we pop all its edges from the stack and assign them labels. Detailed code can be found for example in [Sch98a].