20 Biconnected components

Biconnectivity in an undirected graph is based on a relation between edges, just as strong connectivity was based on a relation between vertices.

(20.1) Definition *Let G be an (undirected) graph. A* simple cycle *in G is a cycle in the bidirected graph, containing at least 3 vertices and edges, in which every vertex occurs just once (except that the first and last vertices are the same).*

Two edges of G[1] *are* biconnected *if either they are the same edge, or there exists a simple cycle containing both.*

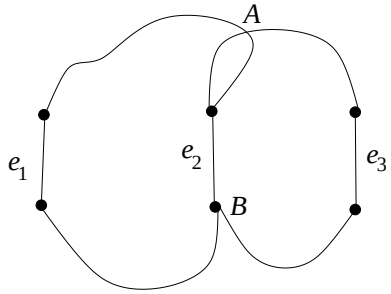
(20.2) Lemma *This is an equivalence relation on edges.*

Proof — sketch. The relation is clearly reflexive and symmetric. For transitivity, let us use $\sim$ to denote the relation. Suppose that $e_1 \sim e_2$ and $e_2 \sim e_3$.

Either $e_1 = e_2$ or there exists a simple cycle C_1 containing e_1 and e_2. Either $e_2 = e_3$ or there exists a simple cycle C_2 containing e_2 and e_3. We want a simple cycle C_3 containing e_1 and e_3. If $e_1 = e_3$ or $e_2 = e_3$ then $e_1 \sim e_3$.

Otherwise, if $e_1 \in C_2$ take $C_3 = C_2$ and if $e_3 \in C_1$ take $C_3 = C_1$.

So we suppose that C_1 does not contain e_3 nor does C_2 contain e_1.



Follow the cycle C_1 clockwise from e_1 until it meets a vertex A in C_2. This happens because $e_2 \in C_1 \cap C_2$. Follow the cycle C_1 anticlockwise from e_1 until it meets a vertex B in C_2.

There is a path P within C_1 containing e_1 and joining the two vertices A and B.

The vertices A, B split C_2 into two paths, one, Q, containing e_3. Then $P \cup Q$ is a simple cycle containing e_1 and e_3. ∎

(20.3) Definition *A* biconnected component *is the subgraph spanned by one of the equivalence classes of edges under the 'biconnected' relation.*

Biconnected components are edge-disjoint but not always vertex-disjoint.

20.1 Articulation points

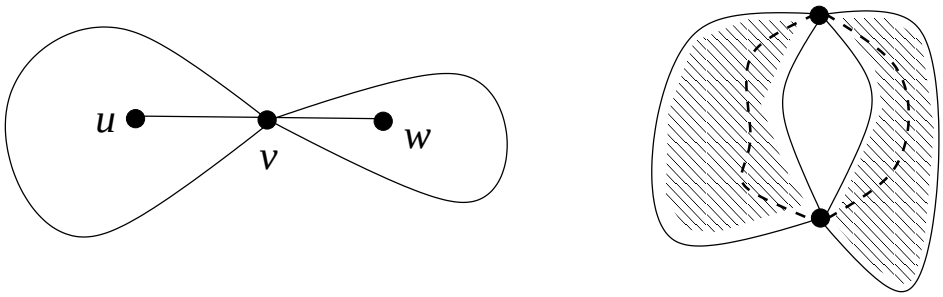
Let G be a graph, for simplicity, a connected graph.[2]

An *articulation point* v in G is a vertex such that $G \backslash v$ is *disconnected.* Equivalently: there exist two edges $\{u, v\}$ and $\{v, w\}$ such that every path from u to w must pass through v.

[1] Undirected edges, not pairs of inverse edges

[2] Meaning that every vertex is reachable from every other vertex

(20.4) Lemma *Any vertex common to two BCCs is an articulation point. Two BCCs can intersect in at most one vertex, which is an articulation point. (Proof: see figure).* ∎



20.2 Tree edges, back edges, and highpt

Suppose an undirected (or rather bidirected) graph is subjected to a full depth-first-search.

(20.5) Definition *Recall that for every undirected edge $\{u, v\}$ of the graph there are two directed edges (u, v) and (v, u) in the bidirected graph.*

Following a full dfs, an edge (u, v) is

- *A* tree edge *if u is the parent of v*
- *An* inverse tree edge *if v is the parent of u*
- *A* forward edge *if (u, v) is not a tree edge but v is a descendant of u, and*
- *A* back edge *if (v, u) is a forward edge.*

(20.6) Lemma *The above list is exhaustive.*

Proof. Suppose that (u, v) is an edge and v follows u in preorder. By the depth-first property, v is a descendant of u, so (u, v) is either a tree edge or a forward edge.

Suppose that u follows v in preorder. Since (v, u) is another edge of G, it is either a tree edge or a forward edge and (u, v) is either an inverse tree edge or a back edge. ∎

After the dfs, which produces `parent` and `pre_rank` functions, the directed edges (u, v) can be classified as follows:

- Tree edge: u is the parent of v
- Back edge: `pre_rank[`u`] > pre_rank[`v`]` **and** $v \neq$ `parent[u]`.
- Otherwise: a forward edge or an inverse tree edge.

(20.7) Definition `highpt(u)` *is the 'highest' vertex v either equal to u or reachable from u by a tree branch followed by a single back edge. 'Highest' means that its* `pre_rank` *is minimal.*

Biconnected components can be identified using a modified depth-first search technique.

The routine will store edges in a pushdown stack at the time they are 'seen' as tree- or back-edges, and periodically 'popped' from the stack when a BCC has been identified.

The criterion is simple: Given a BCC B, if $\{u, v\}$ is the first edge in B seen in dfs, directed as (u, v), say, then it must be a tree edge: `dfs(v)` is called within `dfs(u)`, and when `dfs(v)` ends, `highpt[v]` will follow `highpt[u]` in preorder. This is a signal for the stack to be 'popped' back to (u, v).

The reason is plausible: If u is not the root, with parent w, say, then all the tree edges (u, v) are in different BCCs, and they are in BCCs which differ from that containing $\{u, w\}$ (and u is an articulation point). If u is the root, then all tree edges (u, v) from u belong to different BCCs (the root is an articulation point if and only if it has more than one DFS child).

```
void dfs (u) // 'pseudocode'
{
  set pre_rank[u];
  set highpt[u] = u;
  for all edges (u,v) out of u
  if ( u,v ) is not forward and not inverse tree
  {
    push (u,v);
    if ( (u,v) is a back-edge
        and v precedes highpt[u] in preorder
        set highpt[u] = v;

    if ( v has not yet been visited )
    {  parent[v] = u; // (u,v) a tree edge
       dfs (v);

       if ( highpt[v] precedes highpt[u] in preorder )
          highpt[u] = highpt[v]
       else
       { printf ("New BCC");
         repeat
            EDGE * e = pop(stack);
            print edge e;
         until
            e == (u,v);
       }
    }
  }
}
```