6.2.5 Biconnected Components

The operations that we have implemented thus far are simple extensions of depth first and breadth first search. The next operation we implement is more complex and requires the introduction of additional terminology. We begin by assuming that G is an undirected connected graph.

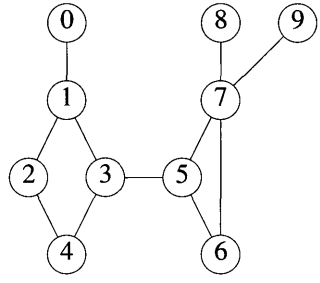
An *articulation point* is a vertex v of G such that the deletion of v, together with all edges incident on v, produces a graph, G', that has at least two connected components. For example, the connected graph of Figure 6.19 has four articulation points, vertices 1, 3, 5, and 7.

A *biconnected graph* is a connected graph that has no articulation points. For example, the graph of Figure 6.16 is biconnected, while the graph of Figure 6.19 obviously is not. In many graph applications, articulation points are undesirable. For instance, suppose that the graph of Figure 6.19(a) represents a communication network. In such graphs, the vertices represent communication stations and the edges represent communication links. Now suppose that one of the stations that is an articulation point fails. The result is a loss of communication not just to and from that single station, but also between certain other pairs of stations.

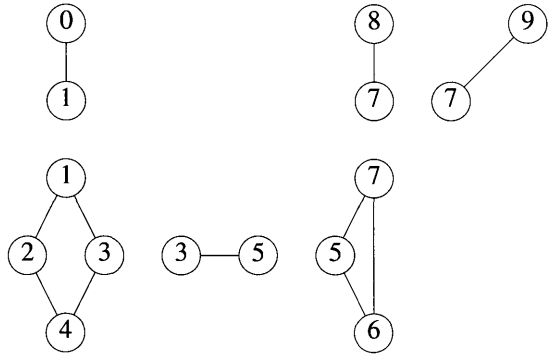
A *biconnected component* of a connected undirected graph is a *maximal biconnected subgraph*, H, of G. By maximal, we mean that G contains no other subgraph that is both biconnected and properly contains H. For example, the graph of Figure 6.19(a) contains the six biconnected components shown in Figure 6.19(b). The biconnected graph of Figure 6.16, however, contains just one biconnected component: the whole graph. It is easy to verify that two biconnected components of the same graph have no more than one vertex in common. This means that no edge can be in two or more biconnected components of a graph. Hence, the biconnected components of G partition the edges of G.

We can find the biconnected components of a connected undirected graph, G, by using any depth first spanning tree of G. For example, the function call *dfs* (3) applied to the graph of Figure 6.19(a) produces the spanning tree of Figure 6.20(a). We have redrawn the tree in Figure 6.20(b) to better reveal its tree structure. The numbers outside the vertices in either figure give the sequence in which the vertices are visited during the depth first search. We call this number the *depth first number*, or *dfn*, of the vertex. For example, $dfn\,(3) = 0$, $dfn\,(0) = 4$, and $dfn\,(9) = 8$. Notice that vertex 3, which is an ancestor of both vertices 0 and 9, has a lower *dfn* than either of these vertices. Generally, if u and v are two vertices, and u is an ancestor of v in the depth first spanning tree, then $dfn\,(u) < dfn\,(v)$.

The broken lines in Figure 6.20(b) represent nontree edges. A nontree edge (u, v) is a *back edge iff* either u is an ancestor of v or v is an ancestor of u. From the definition of depth first search, it follows that all nontree edges are back edges. This means that the root of a depth first spanning tree is an articulation point *iff* it has at least two children. In addition, any other vertex u is an articulation point *iff* it has at least one child w such



(a) Connected graph



(b) Biconnected components

Figure 6.19: A connected graph and its biconnected components

that we cannot reach an ancestor of u using a path that consists of only w, descendants of w, and a single back edge. These observations lead us to define a value, *low*, for each vertex of G such that $low(u)$ is the lowest depth first number that we can reach from u using a path of descendants followed by at most one back edge:

$$low(u) = \min\{dfn(u), \min\{low(w) \mid w \text{ is a child of } u\}, \min\{dfn(w) \mid (u, w) \text{ is a back edge}\}\}$$

Therefore, we can say that u is an articulation point *iff* u is either the root of the spanning tree and has two or more children, or u is not the root and u has a child w such that $low(w) \geq dfn(u)$. Figure 6.21 shows the *dfn* and low values for each vertex of the spanning tree of Figure 6.20(b). From this table we can conclude that vertex 1 is an

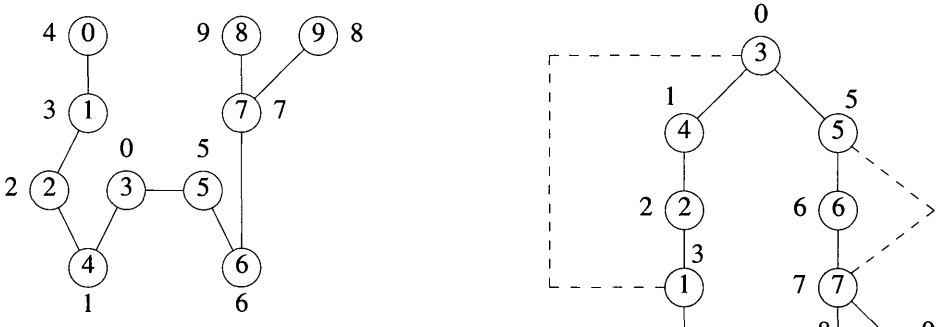


Figure 6.20: Depth first spanning tree of Figure 6.19(a)

articulation point since it has a child 0 such that $low(0) = 4 \geq dfn(1) = 3$. Vertex 7 is also an articulation point since $low(8) = 9 \geq dfn(7) = 7$, as is vertex 5 since $low(6) = 5 \geq dfn(5) = 5$. Finally, we note that the root, vertex 3, is an articulation point because it has more than one child.

Vertex	0	1	2	3	4	5	6	7	8	9
dfn	4	3	2	0	1	5	6	7	9	8
low	4	3	0	0	0	5	5	7	9	8

Figure 6.21: *dfn* and *low* values for *dfs* spanning tree with *root* = 3

We can easily modify *dfs* to compute *dfn* and *low* for each vertex of a connected undirected graph. The result is *dfnlow* (Program 6.4).

We invoke the function with the call *dfnlow(x, −1)*, where x is the starting vertex for the depth first search. The function uses a *MIN*2 macro that returns the smaller of its two parameters. The results are returned as two global variables, *dfn* and *low*. We also use a global variable, *num*, to increment *dfn* and *low*. The function *init* (Program 6.5) contains the code to correctly initialize *dfn*, *low*, and *num*. The global declarations are:

```
void dfnlow(int u, int v)
{/* compute dfn and low while performing a dfs search
   beginning at vertex u, v is the parent of u (if any) */
  nodePointer ptr;
  int w;
  dfn[u] = low[u] = num++;
  for (ptr = graph[u]; ptr; ptr = ptr→link) {
    w = ptr→vertex;
    if (dfn[w] < 0) { /* w is an unvisited vertex */
      dfnlow(w,u);
      low[u] = MIN2(low[u],low[w]);
    }
    else if (w != v)
      low[u] = MIN2(low[u],dfn[w]);
  }
}
```

Program 6.4: Determining *dfn* and *low*

```
#define MIN2(x,y) ((x) < (y) ? (x) : (y))
short int dfn[MAX_VERTICES];
short int low[MAX_VERTICES];
int num;
```

```
void init(void)
{
  int i;
  for (i = 0; i < n; i++) {
    visited[i] = FALSE;
    dfn[i] = low[i] = -1;
  }
  num = 0;
}
```

Program 6.5: Initialization of *dfn* and *low*

We can partition the edges of the connected graph into their biconnected

components by adding some code to *dfnlow*. We know that *low[w]* has been computed following the return from the function call *dfnlow* (*w*, *u*). If *low* [*w*] ≥ *dfn* [*u*], then we have identified a new biconnected component. We can output all edges in a biconnected component if we use a stack to save the edges when we first encounter them. The function *bicon* (Program 6.6) contains the code. The same initialization function (Program 6.5) is used. The function call is *bicon* (*x*, −1), where *x* is the root of the spanning tree. Note that the parameters for the stack operations *push* and *pop* are slightly different from those used in Chapter 3.

```
void bicon(int u, int v)
{/* compute dfn and low, and output the edges of G by their
    biconnected components, v is the parent (if any) of u
    in the resulting spanning tree. It is assumed that all
    entries of dfn[] have been initialized to -1, num is
    initially to 0, and the stack is initially empty */
   nodePointer ptr;
   int w,x,y;
   dfn[u] = low[u] = num++;
   for (ptr = graph[u]; ptr; ptr = ptr→link) {
      w = ptr→vertex;
      if (v != w && dfn[w] < dfn[u])
         push(u,w); /* add edge to stack */
         if (dfn[w] <0) { /* w has not been visited */
            bicon(w,u);
            low[u] = MIN2(low[u],low[w]);
            if (low[w] >= dfn[u]) {
               printf("New biconnected component: ");
               do { /* delete edge from stack */
                  pop(&x, &y);
                  printf(" <%d,%d>",x,y);
               } while (!((x == u) && (y == w)));
               printf("\n");
            }
         }
         else if (w != v) low[u] = MIN2(low[u],dfn[w]);
   }
}
```

Program 6.6: Biconnected components of a graph

Analysis of *bicon*: The function *bicon* assumes that the connected graph has at least two vertices. Technically, a graph with one vertex and no edges is biconnected, but, our implementation does not handle this special case. The complexity of *bicon* is $O(n + e)$. We leave the proof of its correctness as an exercise. □

EXERCISES

1. Rewrite *dfs* so that it uses an adjacency matrix representation of graphs.
2. Rewrite *bfs* so that it uses an adjacency matrix representation.
3. Let G be a connected undirected graph. Show that no edge of G can be in two or more biconnected components of G. Can a vertex of G be in more than one biconnected component?
4. Let G be a connected graph and let T be any of its depth first spanning trees. Show that every edge of G that is not in T is a back edge relative to T.
5. Write the stack operations necessary to fully implement the *bicon* function. Use a dynamically linked representation for the stack.
6. Prove that function *bicon* correctly partitions the edges of a connected graph into the biconnected components of the graph.
7. A *bipartite graph*, $G = (V, E)$, is an undirected graph whose vertices can be partitioned into two disjoint sets V_1 and $V_2 = V - V_1$ with the properties:

   - no two vertices in V_1 are adjacent in G
   - no two vertices in V_2 are adjacent in G

   The graph G_4 of Figure 6.5 is bipartite. A possible partitioning of V is $V_1 = \{0, 3, 4, 6\}$ and $V_2 = \{1, 2, 5, 7\}$. Write a function to determine whether a graph is bipartite. If the graph is bipartite your function should obtain a partitioning of the vertices into two disjoint sets, V_1 and V_2, satisfying the two properties listed. Show that if G is represented by its adjacency lists, then this function has a computing time of $O(n + e)$, where $n = |\ V(G)\ |$ and $e = |\ E(G)\ |$ ($|\ |$ is the cardinality of the set, that is, the number of elements in it).
8. Show that every tree is a bipartite graph.
9. Prove that a graph is bipartite *iff* it contains no cycles of odd length.
10. Apply depth first and breadth first searches to the complete graph on four vertices. List the vertices in the order that they are visited.
11. Show how to modify *dfs* as it is used in *connected* to produce a list of all newly visited vertices.