4.4.4 Depth-first Search Trees

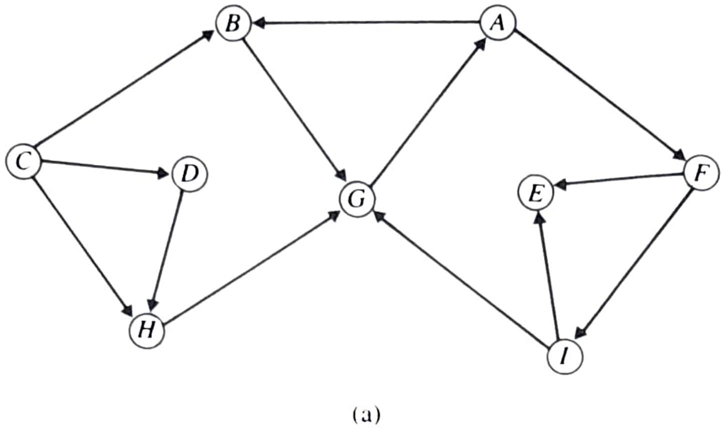
The edges that lead to new, i.e., unmarked, vertices during a depth-first search of a graph or digraph G form a rooted tree called a depth-first search tree. If not all of the vertices can be reached from the starting vertex (the root), then a complete traversal of G partitions the vertices into several trees. For an undirected graph, the search provides an orientation for each of its edges; they are oriented in the direction in which they are traversed. (If G is directed, its edges may be traversed only in the direction of their preassigned orientation.)

We say that a vertex v is an *ancestor* of a vertex w in a tree if v is on the path from the root to w; v is a proper ancestor of w if $v \neq w$. If v is a (proper) ancestor of w, then w is a (proper) descendant of v.

An edge of G that is traversed from a vertex to one of its ancestors in a depth-first search tree is called a *back edge*. If G is undirected, each of its edges will be a tree edge or a back edge. If G is a digraph, depth-first search partitions its edges into several classes: tree edges, back edges, edges that go from a vertex to one of its descendants other than a child, and edges called *cross edges* between two vertices such that neither is a descendant of the other. See Fig. 4.24 for illustrations. Note that the head and tail of a cross edge may be in two different trees. The reader should prove that there can be no cross edges or descendant edges if G is undirected. The distinctions between the various types of edges are important in some applications of depth-first search — in particular, in the algorithms studied in Sections 4.5 and 4.6.

4.4.5 A Generalized Depth-first Search Skeleton

Depth-first search provides the structure for many elegant and efficient algorithms. A depth-first search encounters each vertex several times: when the vertex is first visited and becomes part of the depth-first search tree, then several more times when the search backs up *to* it and attempts to branch out in a different direction, and finally, after the last of these encounters, when the search backs up *from* the vertex and does not pass through it or any of its descendants again. Depending on the



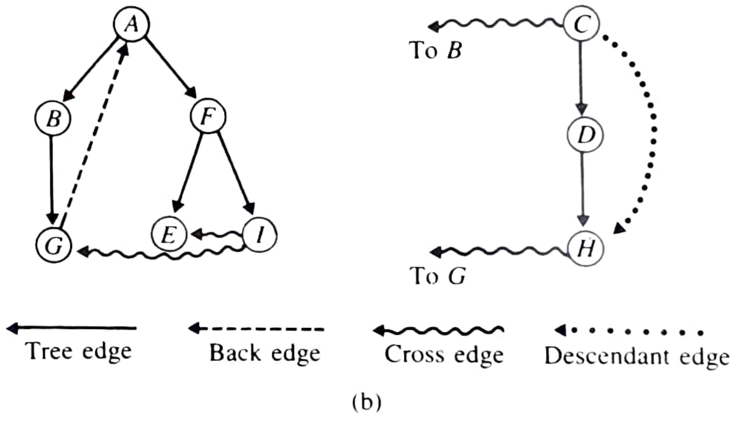


Figure 4.24 (a) A digraph. (b) Depth-first search trees for the digraph.

problem to be solved, an algorithm will process the vertices differently when they are encountered at various stages of the traversal. Many algorithms will also do some computation for the edges: perhaps for each edge, or perhaps only for edges in the depth-first search tree, or perhaps different kinds of computation for the different kinds of edges. The following skeleton algorithm shows exactly where the processing would be done for each kind of edge and for each kind of encounter with the vertices.

Algorithm 4.8 General Depth-first Search Skeleton

Input: $G = (V, E)$, a graph or digraph represented by the adjacency list structure described in Section 4.1.2 with $V = \{1, 2, \ldots, n\}$.

```
var
    mark: array[VertexType] of integer;
    markValue: integer;

procedure DFS (v: VertexType);
{ Does a depth-first search beginning at the vertex v, marking
  the vertices with markValue. }
```

```
var
    w: VertexType;
    ptr: NodePointer;

begin
    { Process vertex when first encountered (like preorder). }
    mark[v] := markValue;

    ptr := adjacencyList[v];
    while ptr ≠ nil do
        w := ptr↑.vertex;

        { Processing for every edge.
          (If G is undirected, each edge is encountered
          twice; an algorithm may have to distinguish the
          two encounters.)                              }

        if mark[w] = 0 { unmarked } then

            { Processing for tree edges, vw. }

            DFS(w);

            { Processing when backing up to v (like inorder) }

        else

            { Processing for nontree edges.
              (If G is undirected, an algorithm may have
              to distinguish the case where w is the
              parent of v.)                          }

        end { if };
        ptr := ptr↑.link
    end { while };

    { Processing when backing up from v (like postorder) }

end { DFS }
```

For an exercise, the reader should write an algorithm to determine if a graph (undirected) has a cycle. (This will require distinguishing between a nontree edge and an encounter with a tree edge "backwards," i.e., from a vertex to its parent.)

In some applications of depth-first search, we may need to know which vertices are on the path from the root to the current vertex (v). They are exactly the vertices that are on the stack. For some algorithms we need to know the order in which vertices are encountered for the first time. We can simply number the vertices as they are encountered by incrementing *markValue*. The number assigned to a vertex in this way is called its *depth-first search number*.

4.5 Biconnected Components of a Graph

4.5.1 Articulation Points and Biconnected Components

In Section 4.1 we raised these questions:

If one city's airport is closed by bad weather, can you still fly between any other pair of cities?

If one computer in a network goes down, can messages be sent between any other pair of computers in the network?

In this section we consider undirected graphs only. As a graph problem, the question is:

If any one vertex (and the edges incident with it) is removed from a connected graph, is the remaining subgraph still connected?

This question is important in graphs representing all kinds of communication or transportation networks. It is also important to find those vertices, if any, whose removal can disconnect the graph.

Formally, a vertex v is an *articulation point* (also called a *cutpoint*) for a graph if there are distinct vertices w and x (distinct from v also) such that v is in every path from w to x. Clearly, the removal of an articulation point would leave an unconnected graph, so a connected graph is *biconnected* if and only if it has no articulation points. A *biconnected component* of a graph is a maximal biconnected subgraph, that is, a biconnected subgraph not contained in any larger biconnected subgraph. Figure 4.25 illustrates biconnected components. Observe that, although the biconnected components partition the edges into disjoint sets, they do not partition the vertices; some vertices are in more than one component. (Which vertices are these?)

There is an alternative characterization of biconnected components, in terms of an equivalence relation on the edges, that is sometimes useful. Two edges e and e' are equivalent if $e = e'$ or if there is a cycle containing both e and e'. Then each subgraph consisting of the edges in one equivalence class and the incident vertices is a biconnected component. (Verifying that the relation described is indeed an equivalence relation and verifying that it characterizes the biconnected components are left as exercises.)

The applications that motivate the study of biconnectivity should suggest a dual problem to the reader: how to determine if there is an edge whose removal would disconnect a graph, and how to find such an edge if there is one. For example, if a railroad track is damaged, can trains still travel between any pair of stations? Relationships between the two problems are examined in Exercise 4.40.

The algorithm we will study for finding biconnected components uses the depth-first search skeleton of Section 4.4.5 and the idea of a depth-first search tree from Section 4.4.4. During the search, information will be computed and saved so

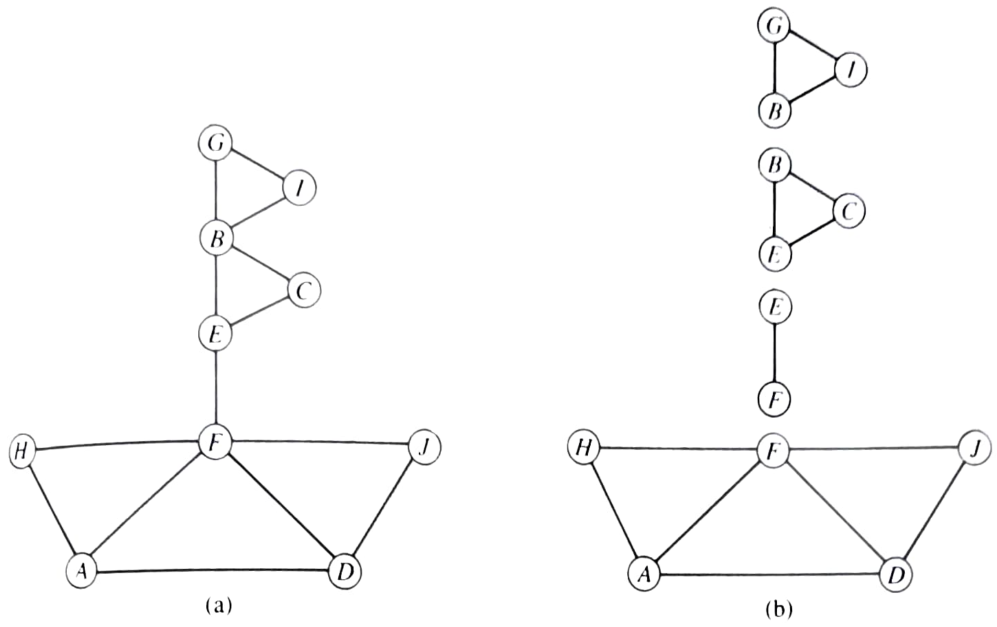


Figure 4.25 (a) A graph. (b) Its biconnected components.

that the edges (and, implicitly, the incident vertices) can be divided into biconnected components as the search progresses. What information must be saved? How is it used to determine the biconnected components? Several wrong answers to these questions seem reasonable until they are examined carefully. Two edges are in the same component if they are in a cycle, and every cycle must include at least one back edge. The reader should work on Exercise 4.32 before proceeding; it requires looking at a number of examples to determine relationships between back edges and biconnected components.

From now on we will use the shorter term "bicomponent" in place of "biconnected component."

4.5.2 The Bicomponent Algorithm

Processing of vertices may be done when a vertex is *first visited*, when the search backs up *to* it, and/or when the search backs up *from* it. The bicomponent algorithm tests to see if a vertex in the tree is an articulation point each time the search backs up to it. Suppose the search is backing up to v from w. If there is no back edge from any vertex in the subtree rooted at w to a proper ancestor of v, then v must be on every path in G from the root to w and is therefore an articulation point. See Fig. 4.26 for illustration. (The careful reader will note that this argument is not valid if v is the root.) The subtree rooted at w, along with all back edges leading from it and along with the edge vw, can be separated from the rest of the graph at v, but it is not necessarily one bicomponent; it may be a union of several. We ensure that bicomponents are properly separated by removing each one as soon as it is detected. Vertices at the outer extremities of the tree are tested for articulation points before vertices closer to the root, ensuring that when an articulation point is found, the subtree

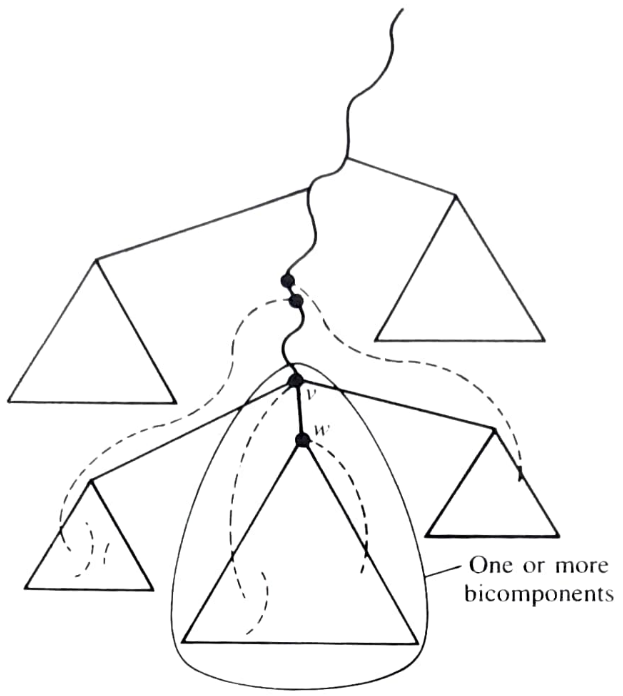


Figure 4.26 An articulation point v in a depth-first search tree. Every path from the root to w passes through v.

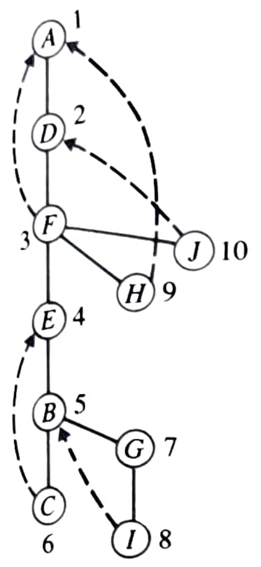
in question (along with the additional edges mentioned above) forms one bicomponent.

This discussion suggests that the algorithm must keep track of how far back in the tree one can get from each vertex by following tree edges (implicitly directed away from the root) and certain back edges. This information will be stored in an array *back*. The vertices will be numbered in the order in which they are first visited. These numbers, stored in an array *dfsNumber*, replace the marks used earlier. Values of *back* will be these vertex numbers. For a vertex v, $back[v]$ may be assigned (or modified) when the search is going forward and a back edge from v is encountered (as in Fig. 4.27(b) with $v = F$ and in Fig. 4.27(c) with $v = C$) and when the search backs up to v (as in Fig. 4.27(d) with $v = B$), since any vertex that can be reached from a child of v can also be reached from v. Determining which of two vertices is farther back in the tree is easy: If v is a proper ancestor of w, then $dfsNumber[v] < dfsNumber[w]$. Thus we can formulate the following rules for setting $back[v]$:

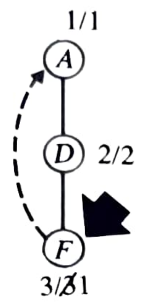
1. When proceeding forward from v and a back edge vw is detected, $back[v] := \min(back[v], dfsNumber[w])$.
2. When backing up from w to v, $back[v] := \min(back[v], back[w])$.

(These rules imply that $back[v]$ must be properly initialized; $back[v]$ will initially be assigned $dfsNumber[v]$, but see Exercise 4.35.)

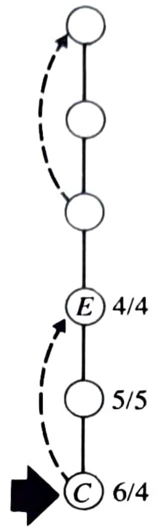
The condition tested to detect a bicomponent when backing up from w to v is $back[w] \geq dfsNumber[v]$. (This condition is tested but not satisfied in Figs. 4.27(d)



(a)
The complete depth-first search tree.

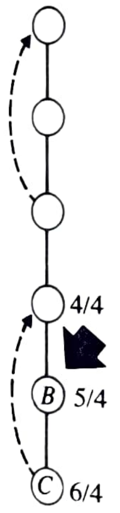


(b)
Proceed forward;
initialize values of *back*.
Detect back edge *FA*;
update *back*[*F*]

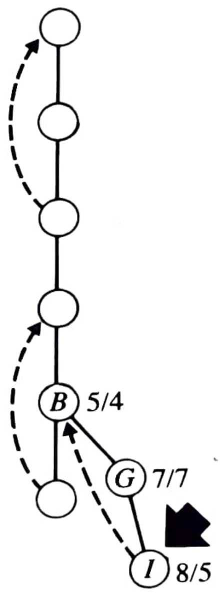


(c)
Continue forward.
Detect back edge *CE*;
update *back*[*C*]

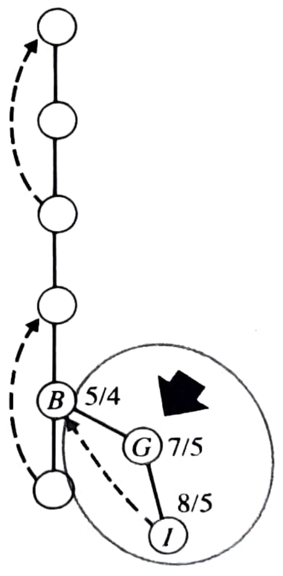
Vertex labels for (b)–(g) are *dfsNumber*/*back*.



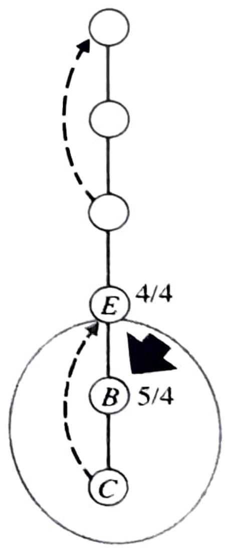
(d)
back[*C*] < *dfsNumber*[*B*] so back up to *B* updating *back*[*B*].



(e)
Forward to *G* and *I*;
detect back edge *IB*;
update *back*[*I*];
back[*I*] < *dfsNumber*[*G*].



(f)
Back up to *G* updating *back*[*G*]; *back*[*G*] = *dfsNumber*[*B*];
remove bicomponent.



(g)
Back up to *B*;
back[*B*] = *dfsNumber*[*E*];
remove bicomponent.

Figure 4.27 The action of the bicomponent algorithm on the graph in Fig. 4.25 (detecting the first two bicomponents).

and 4.27(e); it is satisfied in Figs. 4.27(f) and 4.27(g).) When the test is satisfied, v is an articulation point (except perhaps if v is the root of the tree); a complete bicomponent has been found and may be removed from further consideration. When this occurs, rule 2 above for resetting *back*[v] may be skipped.

The problem of exactly when and how to test for bicomponents is subtle but critical to the correctness of an algorithm. (See Exercises 4.37–4.39.) The essence of the correctness argument is contained in the following theorem.

Theorem 4.3 In a depth-first search tree, a vertex v, other than the root, is an articulation point if and only if v is not a leaf and some subtree of v has no back edge incident with a proper ancestor of v.

Proof. Suppose that v, a vertex other than the root, is an articulation point. Then there are vertices x and y such that v, x, and y are distinct and v is on every path from x to y. At least one of x and y must be a proper descendant of v, since otherwise there would be a path between them using (undirected) edges in the tree without going through v. Thus v is not a leaf. Now suppose every subtree of v has a back edge to a proper ancestor of v; we claim that this contradicts the assumption that v is an articulation point. There are two cases: when only one of x and y is a descendant of v, and when both are descendants of v. For the first case, paths between x and y that do not use v are illustrated in Fig. 4.28. We leave the latter case as an exercise for the reader.

The remaining half of the proof is also left as an exercise. □

Theorem 4.3 does not tell us under what conditions the root is an articulation point. See Exercise 4.34.

We can now outline the work to be done in the depth-first search:

```
procedure BicompDFS(v: VertexType); { outline }
begin
    number v and initialize back[v];
    while there is an untraversed edge vw incident with v do
        if w is unmarked then
            BicompDFS(w);
            { Now backing up to v }
            if back[w] ≥ dfsNumber[v] then
                output a new bicomponent { the subtree rooted at w
                and incident edges };
            else { haven't found a new bicomponent }
                back[v] := min(back[v], back[w])
            end { of backing up from w to v }
        else { w is already in the tree }
            back[v] := min(dfsNumber[w], back[v])
        end { of processing w };
    end { while };
end { BicompDFS }
```

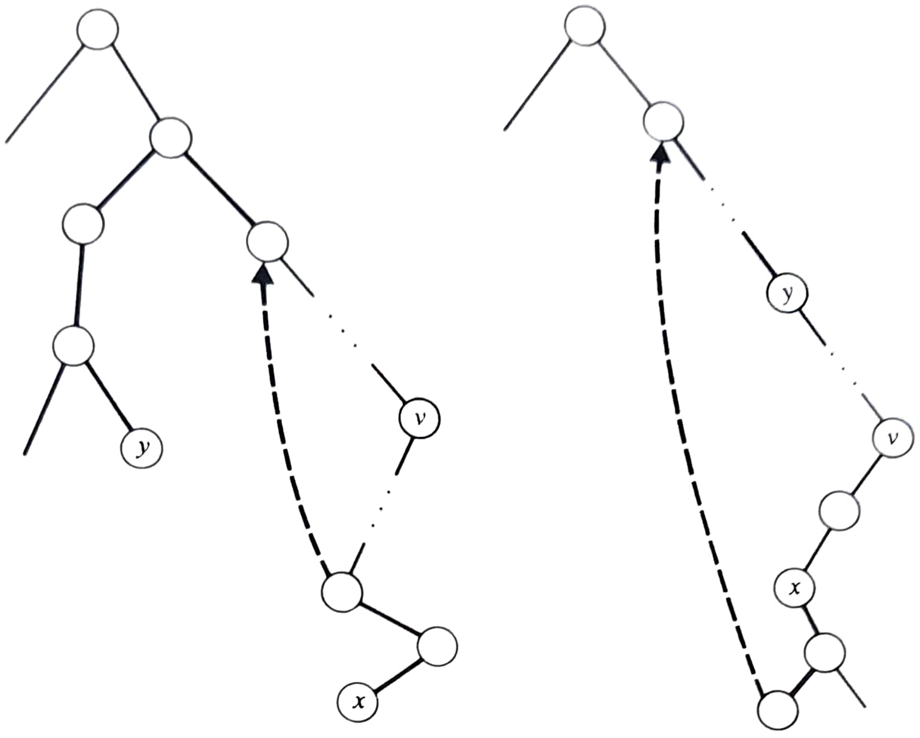


Figure 4.28 Examples for the proof of Theorem 4.3.

The algorithm must keep track of the edges traversed during the search so that those in one bicomponent can easily be identified and removed from further consideration at the appropriate time. As the example in Fig. 4.27 illustrates, when a bicomponent is detected, its edges are the edges most recently processed. Thus edges are stacked on *EdgeStack* as they are encountered. When a bicomponent is detected when backing up from, say, w to v, the edges in that bicomponent are the edges from the top of *EdgeStack* down to (and including) vw. These edges may then be popped.

Each adjacency list will be scanned exactly once, but every edge of G is in two adjacency lists and is encountered twice. Stacking an edge the second time it is encountered can result in incorrect output—some edges may be put in two different bicomponents. The algorithm avoids stacking edges that will cause this problem.

Algorithm 4.9 Biconnected Components

Input: $G = (V, E)$, a connected graph (not directed) represented by linked adjacency lists with $V = \{1, 2, \ldots, n\}$.

Output: Lists of the edges in each biconnected component of G.

```
procedure Bicomponents (adjacencyList: HeaderList; n: integer);
var
    dfsNumber: array[VertexType] of integer;
    back: array[VertexType] of integer;
    dfn: integer;
    v: VertexType;
    EdgeStack: Stack;
    { We assume that Top is a function that returns the top item on a
    stack (without popping it). }

procedure BicompDFS(v: VertexType);
var
    w: VertexType;
    ptr: NodePointer;

begin { BicompDFS }
    { Process vertex when first encountered. }
    dfn := dfn+1;
    dfsNumber[v] := dfn;  back[v] := dfn;
    ptr := adjacencyList[v];
    while ptr ≠ nil do
        w := ptr↑.vertex;
        if dfsNumber[w] < dfsNumber[v] then
            push vw on EdgeStack
            { else wv was a backedge already examined }
        end { if };
        if dfsNumber[w] = 0 { unmarked } then
            BicompDFS(w);
            { Now backing up to v }
            if back[w] ≥ dfsNumber[v] then
                output a heading for a new bicomponent;
                repeat
                    output Top(EdgeStack);
                    pop EdgeStack
                until vw is popped;
            else { haven't found a new bicomponent }
                back[v] := min(back[v], back[w])
            end { of backing up from w to v }
        else { w is already in the tree }
            back[v] := min(dfsNumber[w], back[v])
        end { of processing w };
        ptr := ptr↑.link;
    end { while };
end { BicompDFS }

begin { Bicomponents }
    for v := 1 to n do dfsNumber[v] := 0;
    dfn := 0;
    BicompDFS(1)
end { Bicomponents }
```

4.5.3 Analysis

As usual, $n = |V|$ and $m = |E|$. The initialization in *Bicomponents* includes $\Theta(n)$ operations. *BicompDFS* is the depth-first search skeleton with appropriate processing of vertices and edges added. The depth-first search skeleton takes time in $\Theta(\max(n, m)) = \Theta(m)$. (Since G is connected, $m \geq n-1$.) Thus if the amount of processing for each vertex and edge is bounded by a constant, the complexity of *Bicomponents* is in $\Theta(m)$. It is easy to see that this is the case. The needed observation is nontrivial only when the search backs up from w to v. Sometimes the **repeat** loop popping edges from *EdgeStack* is executed, sometimes not, and the number of edges popped each time varies. But each edge is stacked and popped at most twice (yes, some edges may be stacked when encountered from both directions), so overall the amount of work done is in $\Theta(m)$.

The amount of space used is $\Theta(n+m)$.

4.5.4 Generalizations

The prefix "bi" means "two." Informally speaking, a biconnected graph has two vertex-disjoint paths between any pair of vertices (see Exercise 4.29). We can define triconnectivity (and, in general, k-connectivity) to denote the property of having three (in general, k) vertex-disjoint paths between any pair of vertices. An efficient algorithm that uses depth-first search to find the triconnected components of a graph has been developed (see the notes and references at the end of the chapter), but it is much more complicated than the algorithm for bicomponents.