if and only if \( v \) has some child \( w \) such that \( \text{Low}(w) \geq \text{Num}(v) \). Notice that this condition is always satisfied at the root; hence the need for a special test.

The \( \text{if} \) part of the proof is clear when we examine the articulation points that the algorithm determines, namely, \( C \) and \( D \). \( D \) has a child \( E \), and \( \text{Low}(E) \geq \text{Num}(D) \), since both are 4. Thus, there is only one way for \( E \) to get to any node above \( D \), and that is by going through \( D \). Similarly, \( C \) is an articulation point, because \( \text{Low}(G) \geq \text{Num}(C) \). To prove that this algorithm is correct, one must show that the \( \text{only if} \) part of the assertion is true (that is, this finds \textit{all} articulation points). We leave this as an exercise. As a second example, we show (Figure 9.64) the result of applying this algorithm on the same graph, starting the depth-first search at \( C \).

We close by giving pseudocode to implement this algorithm. We will assume that \texttt{Vertex} contains the data fields \texttt{visited} (initialized to \texttt{false}), \texttt{num}, \texttt{low}, and \texttt{parent}. We will also keep a \texttt{(Graph)} class variable called \texttt{counter}, which is initialized to 1, to assign the preorder traversal numbers, \texttt{num}. We also leave out the easily implemented test for the root.

As we have already stated, this algorithm can be implemented by performing a preorder traversal to compute \texttt{Num} and then a postorder traversal to compute \texttt{Low}. A third traversal can be used to check which vertices satisfy the articulation point criteria. Performing three traversals, however, would be a waste. The first pass is shown in Figure 9.65.

The second and third passes, which are postorder traversals, can be implemented by the code in Figure 9.66. The last \texttt{if} statement handles a special case. If \( w \) is adjacent to
// Assign low; also check for articulation points.
void assignLow( Vertex v )
{
    v.low = v.num; // Rule 1
    for each Vertex w adjacent to v
    {
        if( w.num > v.num ) // Forward edge
        {
            assignLow( w );
            if( w.low >= v.num )
                System.out.println( v + " is an articulation point" );
            v.low = min( v.low, w.low ); // Rule 3
        }
        else
            if( v.parent != w ) // Back edge
                v.low = min( v.low, w.num ); // Rule 2
    }
}

Figure 9.66  Pseudocode to compute Low and to test for articulation points (test for the root is omitted)

void findArt( Vertex v )
{
    v.visited = true;
    v.low = v.num = counter++; // Rule 1
    for each Vertex w adjacent to v
    {
        if( !w.visited ) // Forward edge
        {
            w.parent = v;
            findArt( w );
            if( w.low >= v.num )
                System.out.println( v + " is an articulation point" );
            v.low = min( v.low, w.low ); // Rule 3
        }
        else
            if( v.parent != w ) // Back edge
                v.low = min( v.low, w.num ); // Rule 2
    }
}

Figure 9.67  Testing for articulation points in one depth-first search (test for the root is omitted) (pseudocode)
The algorithm is described by performing a depth-first search on G, always starting with vertex A.

In Figure 9.75, the vertices are shown with their numbers.

9.6.5 Finding Strong Components

One use of depth-first search is to test whether or not a directed graph is acyclic. The routine is left as an exercise.

Is it easy to check if the depth-first search is being performed, and if so, how to do it?
Each of the trees (this is easier to see if you completely ignore all nontree edges) in this depth-first spanning forest forms a strongly connected component. Thus, for our example, the strongly connected components are \{G\}, \{H, I, J\}, \{B, A, C, F\}, \{D\}, \{E\}.

To see why this algorithm works, first note that if two vertices \(v\) and \(w\) are in the same strongly connected component, then there are paths from \(v\) to \(w\) and from \(w\) to \(v\) in the original graph \(G\), and hence also in \(G_r\). Now, if two vertices \(v\) and \(w\) are not in the same depth-first spanning tree of \(G_r\), clearly they cannot be in the same strongly connected component.

To prove that this algorithm works, we must show that if two vertices \(v\) and \(w\) are in the same depth-first spanning tree of \(G_r\), there must be paths from \(v\) to \(w\) and from \(w\) to