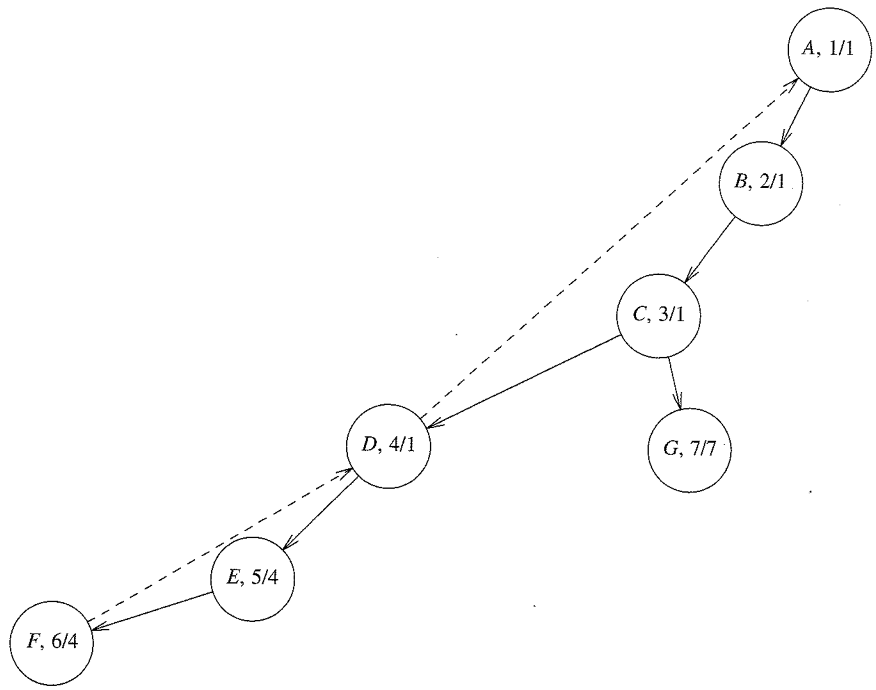

Figure 9.63 Depth-first tree for previous graph, with *Num* and *Low*

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

The *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 $Low(E) \geq Num(D)$, 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 $Low(G) \geq Num(C)$. To prove that this algorithm is correct, one must show that the *only if* part of the assertion is true (that is, this finds *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 `Vertex` contains the data fields `visited` (initialized to `false`), `num`, `low`, and `parent`. We will also keep a (`Graph`) class variable called `counter`, which is initialized to 1, to assign the preorder traversal numbers, `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 *Num* and then a postorder traversal to compute *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 `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)

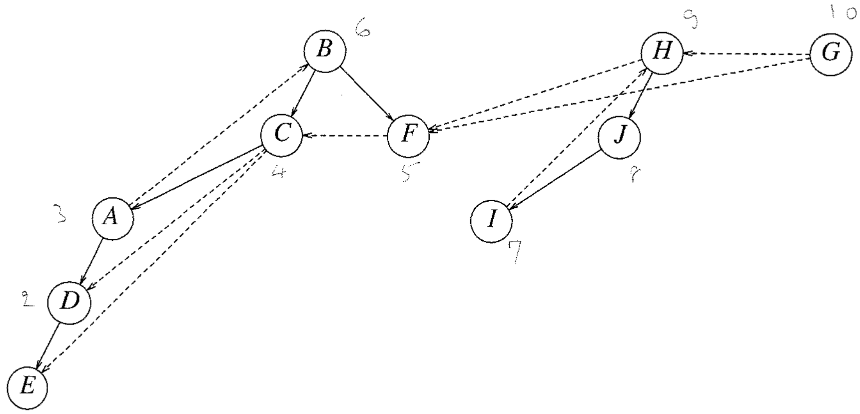

Figure 9.75 Depth-first search of previous graph

left to right. In a depth-first search of a directed graph drawn in this manner, cross edges always go from right to left.

Some algorithms that use depth-first search need to distinguish between the three types of nontree edges. This is easy to check as the depth-first search is being performed, and it is left as an exercise.

One use of depth-first search is to test whether or not a directed graph is acyclic. The rule is that a directed graph is acyclic if and only if it has no back edges. (The graph above has back edges, and thus is not acyclic.) The reader may remember that a topological sort can also be used to determine whether a graph is acyclic. Another way to perform topological sorting is to assign the vertices topological numbers $N, N-1, \ldots, 1$ by postorder traversal of the depth-first spanning forest. As long as the graph is acyclic, this ordering will be consistent.

9.6.5 Finding Strong Components

By performing two depth-first searches, we can test whether a directed graph is strongly connected, and if it is not, we can actually produce the subsets of vertices that are strongly connected to themselves. This can also be done in only one depth-first search, but the method used here is much simpler to understand.

First, a depth-first search is performed on the input graph G. The vertices of G are numbered by a postorder traversal of the depth-first spanning forest, and then all edges in G are reversed, forming G_r. The graph in Figure 9.76 represents G_r for the graph G shown in Figure 9.74; the vertices are shown with their numbers.

The algorithm is completed by performing a depth-first search on G_r, always starting a new depth-first search at the highest-numbered vertex. Thus, we begin the depth-first search of G_r at vertex G, which is numbered 10. This leads nowhere, so the next search is started at H. This call visits I and J. The next call starts at B and visits A, C, and F. The next calls after this are `dfs(D)` and finally `dfs(E)`. The resulting depth-first spanning forest is shown in Figure 9.77.

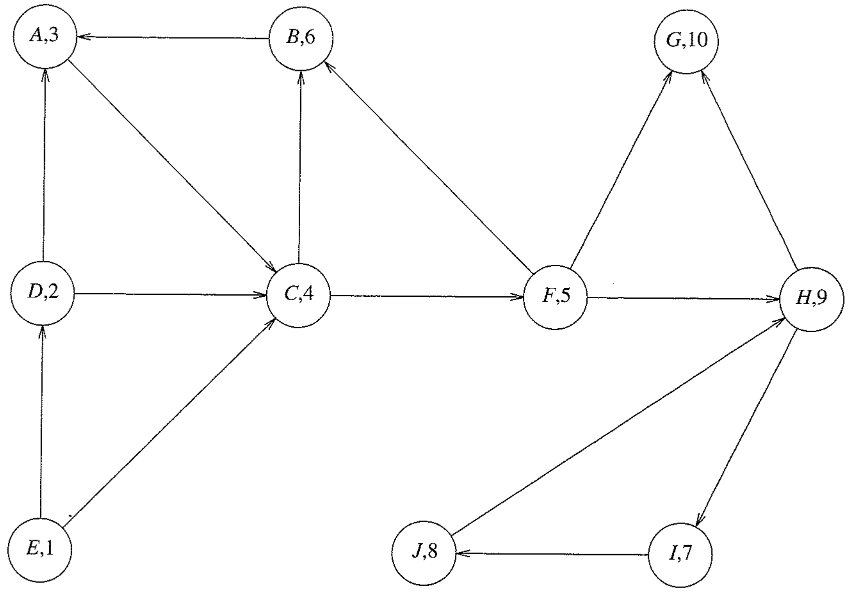

Figure 9.76 G_r numbered by postorder traversal of G (from Figure 9.74)

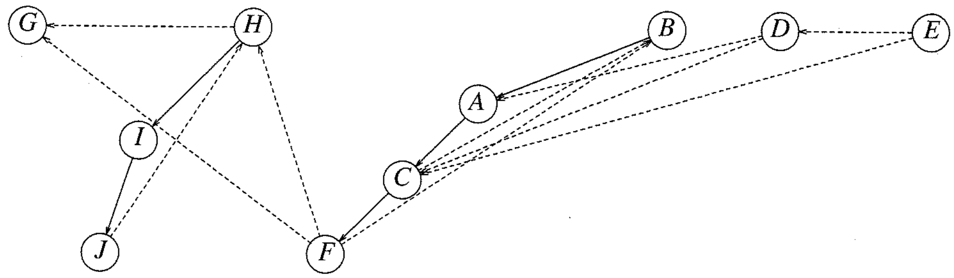

Figure 9.77 Depth-first search of G_r—strong components are $\{G\}$, $\{H, I, J\}$, $\{B, A, C, F\}$, $\{D\}$, $\{E\}$

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 $\{G\}$, $\{H, I, J\}$, $\{B, A, C, F\}$, $\{D\}$, and $\{E\}$.

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