Excerpted from:

Combinatorial Algorithms Theory and Practice

By Reingold, Nievergelt and Deo

8.2.2 Depth-First Search

One of the most natural ways to explore all the vertices and edges of a graph systematically comes from traversing the graph with a backtrack-style search, similar to Algorithm 4.1, which does a depth-first exploration of a tree. *Depth-first search* on an undirected graph $G = (V, E)$ explores the graph as follows. When we are "visiting" a vertex $v \in V$, we follow one of the edges (v, w) incident on v. If the vertex w has been previously visited, we return to v and choose another edge. If the vertex w has not been previously visited, we visit it and apply the process recursively to w. If all the edges incident on v have been thus examined, we go back along the edge (u, v) that led to v and continue exploring the edges incident on u. We are finished when we try to back up from the vertex at which the exploration began. As we shall see here and in later sections, this technique is extremely useful in determining various properties of both undirected and directed graphs.

Figure 8.5 illustrates how depth-first search examines an undirected graph G represented as an adjacency structure. We start with an arbitrarily chosen vertex a. From a we traverse the first edge that we encounter, which is (a, b). Since b is a vertex never visited before, we stay at b and traverse the first untraversed edge encountered at b, which is (b, c). Now at vertex c, the first untraversed edge that we find is (c, a). We traverse (c, a) and find that a has been previously visited. So we return to c, marking the edge (c, a) in some way (as a dashed line in Figure 8.5) to distinguish it from edges like (b, c), which led to new vertices. Back at vertex c, we look for another untraversed edge and traverse the first one that we encounter, say (c, d). Once again, since d is a new vertex, we stay at d and look for an untraversed edge, in this case (d, b). Since b has been previously visited, we mark edge (d, b) with a dashed line and return to d. Next, since there is no untraversed edge left at d, we back up to c. At c we find an untraversed edge (c, e), which we traverse. Since e is a new vertex, we remain at e and look for an untraversed edge at e, which is (e, a). And since a is an old vertex, we mark (e, a) with a dashed line. Now e is completely examined, and so we return to c. At c we find no untraversed edges left; thus we move back to b and finally, back to a. All the edges incident on a have been traversed, which means that we are finished.

As shown in Figure 8.5, the depth-first search converts the original undirected graph G into a digraph $\vec{G}$ by inducing a direction on each edge (the direction of traversal). The numbers shown next to the edges of $\vec{G}$ in Figure 8.5 indicate the order in which the edges were scanned. The edges of $\vec{G}$ that led to new vertices during the traversal, shown as solid lines in Figure 8.5, form a *directed spanning tree* (forest, if the graph is not connected)—that is, a tree T that spans $\vec{G}$ in which there is a distinguished vertex r, called the *root*, such that there is a directed path from r to every vertex in T but no directed path to r from any vertex in T. If there is a path from vertex x to vertex y in a directed spanning tree, we say that x is an *ancestor* of y and, conversely, that y is a *descendant* of x. If the path is not empty (i.e., $x \neq y$), we say that x is a *proper ancestor* of y and y is a *proper descendant* of x.

The directed spanning tree that results from depth-first search on a simple, connected, undirected graph will be referred to as a *DFS-tree*. The edges of $\vec{G}$ not

Given graph G | Adjacency structure of G | Traversed graph $\vec{G}$

Figure 8.5 Depth-first search on a graph.

in the DFS-tree are called the *back edges* because they lead back to vertices that have been previously visited. The back edges in Figure 8.5 are shown with dashed lines. Notice that a back edge must point from a descendant to an ancestor (why?).

To implement depth-first search, we need to distinguish between vertices that have not yet been visited and vertices that have been visited. This can be accomplished by successively numbering the vertices 1 to $|V|$ as we visit them. Initially, we set $num\ (v) = 0$ for all $v \in V$ to indicate that no vertex has been visited; and when we visit a vertex v for the first time, we set $num\ (v)$ to a nonzero value. Such a depth-first search is given as Algorithm 8.2. The recursive procedure $DFS\ (v, u)$ performs depth-first search on the graph $G = (V, E)$ containing v and constructs a directed spanning tree (V, T) for the graph; u is the father of v in the tree. The graph is represented as an adjacency structure, Adj (v) being the set of vertices adjacent to v. The elements of the set T are the edges of the tree, and the elements of the set B are the back edges. If the graph G is not connected, (V, T) will be a forest.

for $x \in V$ **do** $num\ (x) \leftarrow 0$

$i \leftarrow 0$

$T \leftarrow B \leftarrow \varnothing$

for $x \in V$ **do if** $num\ (x) = 0$ **then** $DFS\ (x, 0)$

procedure $DFS\ (v, u)$

$i \leftarrow i + 1$

$num\ (v) \leftarrow i$

for $w \in \text{Adj}\ (v)$ **do if** $num\ (w) = 0$ **then** $\begin{cases} [\![(v, w) \text{ is a tree edge}]\!] \\ T \leftarrow T \cup \{(v, w)\} \\ DFS\ (w, v) \end{cases}$

else if $num\ (w) < num\ (v)$ **and** $w \neq u$

then $\begin{cases} [\![(v, w) \text{ is a back edge}]\!] \\ B \leftarrow B \cup \{(v, w)\} \end{cases}$

return

Algorithm 8.2 Depth-first search on a simple undirected graph.

Notice that if $num\ (w) = 0$, then edge (v, w) is a tree edge, whereas if $num\ (w) \neq 0$, the test to determine if (v, w) is a back edge is "$num\ (w) < num\ (v)$ and $w \neq u$." If $num\ (w) < num\ (v)$, then the vertex w was visited before v. Thus (v, w) is a back edge, provided that it is not the edge in T that was followed from the father of v to v—that is, provided that $w \neq u$.

Since a call to *DFS* is made exactly once for each newly visited vertex, *DFS* is called $|V|$ times. For each call, the amount of work done is proportional to the number of edges incident on that vertex. Therefore the time required for a depth-first search on an undirected graph is $O(|V| + |E|)$.

Connected Components. The simplest use of the depth-first search technique on undirected graphs is to determine the connected components of an undirected graph $G = (V, E)$. Algorithm 8.3 assigns a unique component number, *compnum* (v), to every vertex w belonging to the component in which the vertex v occurs.

for $v \in V$ **do** *compnum* $(v) \leftarrow 0$

$c \leftarrow 0$

for $v \in V$ **do if** *compnum* $(v) = 0$ **then** $\begin{cases} c \leftarrow c + 1 \\ COMP\ (v) \end{cases}$

procedure *COMP* (x)

  compnum $(x) \leftarrow c$

  for $w \in \text{Adj}\ (x)$ **do if** *compnum* $(w) = 0$ **then** *COMP* (w)

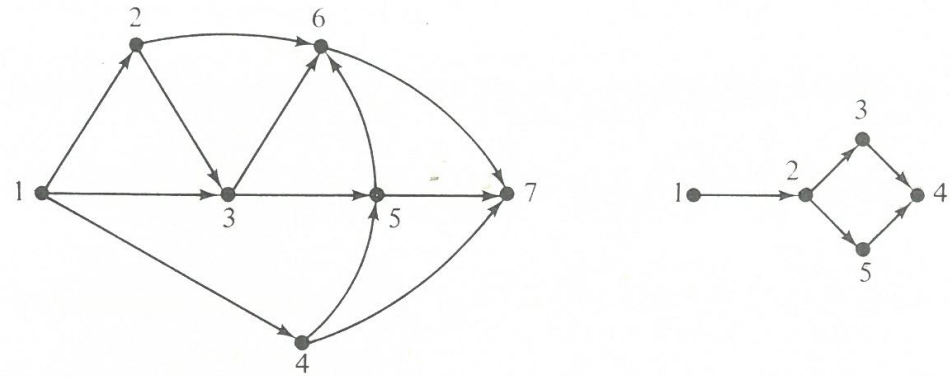
  return

Algorithm 8.3 Determining the connected components of a graph.

It is easy to see that this algorithm, like Algorithm 8.2, requires $O(|V| + |E|)$ operations.

Topological Sorting. The simplest use of the depth-first search technique on digraphs is to determine a labeling of the vertices of an acyclic digraph $G = (V, E)$ with integers $1, 2, \ldots, |V|$, such that if there is a directed edge from vertex i to vertex j, then $i < j$; such a labeling is called a *topological sort* of the vertices of G. For example, the vertices of the digraph in Figure 8.6(a) are topologically sorted but those of Figure 8.6(b) are not. Topological sorting can be viewed as the process of finding a linear order in which a given partial order can be embedded. It is not difficult to show that it is possible to topologically sort the vertices of a digraph if and only if it is acyclic (Exercise 17). Topological sorting is useful in the analysis of activity networks where a large, complex project is represented as a digraph in which the vertices correspond to the goals in the project and the edges correspond to activities. The topological sort gives an order in which the goals can be achieved.

Topological sorting begins by finding a vertex of $G = (V, E)$ with no outgoing edge (such a vertex must exist if G is acyclic) and assigning this vertex the highest number—namely, $|V|$. This vertex is deleted from G, along with all



(a) Topologically sorted. (b) Not topologically sorted.

Figure 8.6 Acyclic digraphs.

incoming edges. Since the remaining digraph is also acyclic, we can repeat the process and assign the next highest number, namely $|V| - 1$, to a vertex with no outgoing edges, and so on. To keep the algorithm $O(|V| + |E|)$, we must avoid searching the modified digraph for a vertex with no outgoing edges.

We do so by performing a single depth-first search on the given acyclic digraph G. In addition to the usual array *num*, we will need another array, *label*, of size $|V|$ for recording the topologically sorted vertex labels. That is, if there is an edge (u, v) in G, then *label* $(u) <$ *label* (v). The complete search and labeling procedure *TOPSORT* is given in Algorithm 8.4.

The time required by Algorithm 8.4 is $O(|V| + |E|)$, since every edge is traversed once and the procedure *TOPSORT* is called once for each vertex.

8.2.3 Biconnectivity

Sometimes it is not enough to know that a graph is connected; we may need to know how "well connected" a connected graph is. A connected graph, for example, may contain a vertex whose removal, along with its incident edges, disconnects the remaining vertices. Such a vertex is called an *articulation point* or a *cut-vertex*. A graph that contains an articulation point is called *separable*. Vertices b, f, and i in Figure 8.7(a), for example, are articulation points, and they are the only ones in the graph. A graph with no articulation points is called *biconnected* or *nonseparable*. A maximal biconnected subgraph of a graph is called a *biconnected component* or a *block*. Identification of the articulation points and biconnected components of a given graph is important in the study of the vulnerability of communication and transportation networks. It is also important in determining other properties, like planarity, of a graph G, since it is often advantageous to separate G into its biconnected components and examine each one individually (see Section 8.6).

A vertex v in an undirected connected graph is an articulation point if and only if there exist two other vertices x and y such that every path between x and y passes through v; in this case and only in this case does the deletion of v from G destroy all paths between x and y (i.e., disconnect G). This observation allows us

```
for x ∈ V do { num (x) ← 0
             { label (x) ← 0
j ← |V| + 1
i ← 0
for x ∈ V do if num (x) = 0 then { ⟦x has no labeled ancestor⟧
                                 { TOPSORT (x)
procedure TOPSORT (v)
    i ← i + 1
    num (v) ← i
    for w ∈ Adj (v) do { ⟦process all descendants of w⟧
                       { if num (w) = 0 then TOPSORT (w)
                       {                 else if label (w) = 0 then ⟦G has a cycle⟧
    ⟦v is labeled with a value less than
     the value assigned to any descendant⟧
    j ← j − 1
    label (v) ← j
    return
```

Algorithm 8.4 Topological sort of an acyclic digraph $G = (V, E)$.

to use depth-first search to find the articulation points and biconnected components of a graph in $O(|V| + |E|)$ operations.

The central idea can be understood by studying the example in Figure 8.8, which shows schematically a connected graph consisting of biconnected components G_i, $1 \leqq i \leqq 9$, and articulation points v_j, $1 \leqq j \leqq 5$. If we start the depth-first search at, say, the vertex s in G_9, we might, perhaps, leave G_9 to go into G_4 by

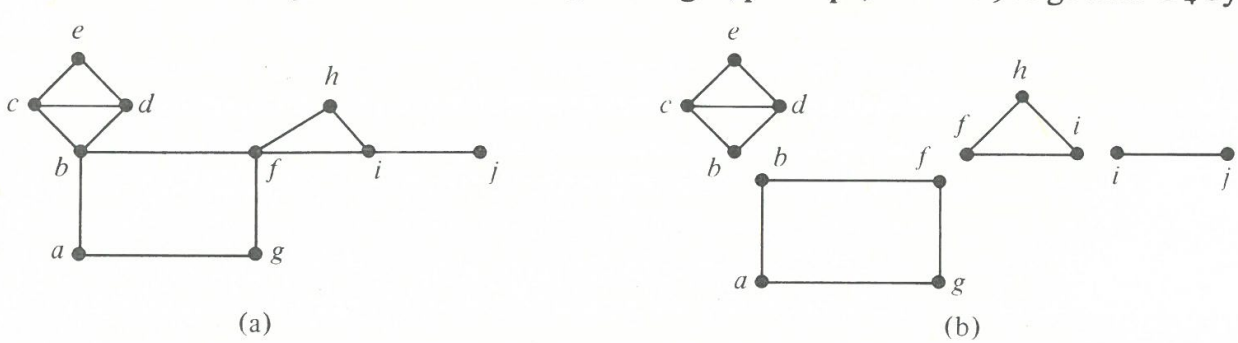


Figure 8.7 A separable graph (a) and its biconnected components (b). The articulation points are *b*, *f*, and *i*.

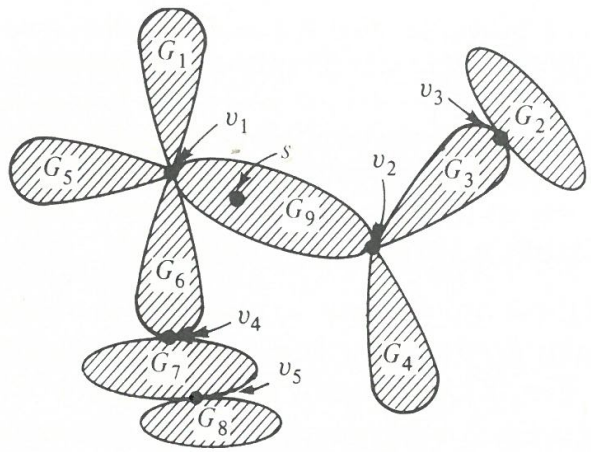


Figure 8.8 A schematic drawing of a graph with nine biconnected components and five articulation points.

passing through v_2. But by the depth-first nature of the search, all the edges in G_4 must be traversed before we back up to v_2; thus G_4 consists of exactly the edges traversed between visits to v_2. Matters are actually a little more complicated for the other biconnected components, since, for example, if we leave G_4 and go into G_3 and from there into G_2 through v_3, we would find ourselves in G_2, having traversed edges from G_9, G_3, and G_2. Fortunately, however, if we store the edges in a stack, by the time we pass through v_3 back into G_3 all the edges of G_2 will be on top of the stack. When they are removed, the edges on top of the stack will be from G_3, and we will once again be traversing G_3. Thus if we can recognize the articulation points, we can determine the biconnected components by applying depth-first search and storing the edges on a stack as they are traversed; the edges on top of the stack as we back up through an articulation point form a biconnected component.

In order to recognize an articulation point we need to compute, during the depth-first search, a new function *lowpt* (v) for every vertex v in the graph. We define *lowpt* (v) as the smallest value of *num* (x), where x is a vertex of the graph that can be reached from v by following a sequence of zero or more tree edges followed by at most one back edge. The function *lowpt* (v) is useful because of the following theorem.

Theorem 8.1

Let $G = (V, E)$ be a connected graph with a DFS-tree T and with back edges B. Then $a \in V$ is an articulation point if and only if there exist vertices v, $w \in V$ such that $(a, v) \in T$, w is not a descendant of v in T and *lowpt* $(v) \geqq$ *num* (a).

Proof: Suppose that such vertices v and w exist. Since $(a, v) \in T$ and *lowpt* $(v) \geqq$ *num* (a), any path starting at v that does not go through a must remain in the subtree with root v. Since w is not a descendant of v in T, such a path cannot contain w. Thus the only paths from v to w contain a and so a is an articulation point.

Conversely, suppose that a is an articulation point. If a is the root of T, then at least two edges of T start at a; otherwise there would be a path in G between every pair of vertices in $V - \{a\}$ that did not contain a. Let (a, v) and (a, w) be two of these edges; clearly, v and w satisfy the theorem. If a is not the root of T, then it has an ancestor w. One of the biconnected components containing a has all its nodes as descendants of a in T; in fact, they are all (except a) descendants of a vertex v; where (a, v) is an edge in T (why?). Clearly, v and w satisfy the theorem.

This theorem tells how to recognize articulation points if we have the values of *num* and *lowpt*: if we find a vertex v such that $(a, v) \in T$ and $lowpt\,(v) \geqq num\,(a)$, then a is either an articulation point or the root of T. This result follows from Theorem 8.1 by observing that a suitable w can be chosen among the ancestors of a if a is not the root. Furthermore, computing the *lowpt* values during depth-first search is simple because

$$lowpt\,(v) = \min\,(\{num\,(v)\} \cup \{lowpt\,(x) | (v, x) \in T\} \cup \{num\,(x) | (v, x) \in B\}).$$

Thus Algorithm 8.5 determines the biconnected components of a graph $G = (V, E)$. Since the algorithm is a depth-first search with a constant amount of extra work done as each edge is traversed, the time required is clearly $O(|V| + |E|)$. A proof that the algorithm works correctly is left as Exercise 20.

Algorithm 8.5 Determining the biconnected components of G = (V,E)

```
i <- 0
S <- empty stack
for v in V do num(v) <- 0
for v in V do if num(v) = 0 then BICON(v,0)

Function BICON(v,u) =
    i++
    num(v) <- i
    lowpt(v) <- i
    for w in Adj(v) do
        if num(w) = 0 then
            // (v,w) is a tree edge
            push (v,w) onto S
            BICON(w,v)
            lowpt(v) <- min (lowpt(v), lowpt(w))
            if lowpt(w) >= num(v) then
                // At this point v is either the root of the tree, or
                // it is an articulation point.
                Form a new biconnected component consisting of
                all the edges on the stack above and including
                (v,w).  Remove these edges from the stack
        else if w = u then
            // do nothing -- (v,w) is a reverse tree edge
        else if num(w) < num(v) then
            // (v,w) is a back edge
            Push (v,w) onto S
            lowpt(v) <- min (lowpt(v), num(w))
        else
            // do nothing -- (v,w) is a reverse back edge
    done
end BICON
```

lowpt(x) is the lowest numbered vertex you can get to from x by following zero or more tree edges (in the down direction), followed by zero or one back edge.

Lemma: The algorithm correctly computes the lowpt() numbers for all vertices.

Theorem: The algorithm computes the bi-connected components correctly.