Kellogg S. Booth

IMPLEMENTING A LINEAR-TIME TEST FOR GRAPH PLANARITY

by

Dolores M. Panek

An essay submitted in partial fulfillment of the requirements for the degree of

Master of Mathematics

University of Waterloo
Department of Computer Science

1978

The writer is grateful to Kelly Booth for patience far beyond any expectation.

1. INTRODUCTION

Several algorithms exist for testing graph planarity, but two stand out as efficient tests. One is a method, presented by Hopcroft and Tarjan [5], which uses depth-first search and achieves a linear running time. The other is an improved, linear time version of a test originally published by Lempel, Even, and Cederbaum [6]. Their method centres around the generation and manipulation of formulas, where each formula represents a particular graph. By introducing an appropriate data structure called a PQ-tree, Booth and Lueker [2] have improved the time bound of the original algorithm. Operations are now performed on PQ-trees rather than on formulas, thus speeding up the algorithm to run in linear time. For both the new version and the original algorithm, the input graph must be biconnected. We know from graph theory that a graph is planar iff each of its biconnected components is planar. Also, the vertices of the input graph must be selected in a particular order. This order can be determined by computing a special numbering of the vertices of a biconnected graph G.

LEMMA 1 (Lempel-Even-Cederbaum). G is a biconnected graph with n vertices iff the vertices can be numbered such that 1 and n are adjacent and for any vertex numbered $1 < j < n$, there exist vertices numbered i and k such that $i < j < k$ and both i and k are adjacent to j.

Such a numbering is called an s-t numbering for a biconnected graph G. A proof of the lemma can be found in [6].

A method for computing an s-t numbering has been published by Even and Tarjan [3]. It requires O(n+e) steps for a graph with n vertices and e edges. Part of their method employs a depth-first search technique for exploring a graph and gathering information about its vertices. In particular, a depth-first search algorithm exists for finding biconnected components in O(n+e) steps [1].

This paper presents an implementation of the depth-first search, the s-t numbering, and the PQ-tree algorithms which combine to provide a linear-time planarity test. Section 2 presents some graph-theoretic definitions. Section 3 discusses the three algorithms used in the computation of an s-t numbering: BICONNECT, PATHFINDER, and STNUMBER. Section 4 proves correctness and linearity for these three algorithms. Section 5 describes how the st-numbering algorithm and an implementation by Young[7] of the PQ-tree algorithms are combined into a single planarity test. Section 6 presents conclusions drawn from running these programs.

2. DEFINITIONS

We now introduce the terms that will be used throughout the discussion of the algorithms. The definitions are similar to those found in most texts on graph theory [4].

A graph $G = (V,E)$ consists of a nonempty set of vertices V ($|V| = n$) and a nonempty set of edges E ($|E| = e$). G is undirected, hence its edges are unordered pairs of distinct vertices represented as (v,w). Thus we consider $(v,w) = (w,v)$. We say w is adjacent to v if (v,w) is in E. A graph $G' = (V',E')$ is a subgraph of a graph $G = (V,E)$ if V' _ V and E' _ E. If $V' = V$ then G is a spanning subgraph.

We define a path in G from v_1 to v_n to be a sequence of edges e_i, $1 \leq i < n$, where $e_i = (v_i, v_{i+1})$. A path can be represented by the sequence $v_1, v_2, \ldots, v_n$ of vertices on the path. If the vertices $v_1, v_2, \ldots, v_n$ are distinct, the path is simple. A path is a cycle if $v_1 = v_n$ is the only duplicated vertex on the path. There is a path of no edges from any vertex to itself. This null path is not considered a cycle.

An undirected graph G is connected if there is a path between any pair of vertices. The maximal connected subgraphs of G are vertex-disjoint and are called its connected components. Given any three distinct vertices u, v, w in some connected component of G, if every path from v to w contains u then u is an articulation point. Removing u from

G splits the graph into two or more disconnected parts. If G has no articulation points, then G is <u>biconnected</u>. The maximal biconnected subgraphs of G are edge-disjoint and are called its <u>biconnected</u> <u>components</u>.

A <u>tree</u> T is a connected, directed graph which contains no cycles. A <u>rooted</u> <u>tree</u> (T,r) is a tree with a distinguished vertex r, called the <u>root</u>. Given any vertices v and w, $v \xrightarrow{*} w$ denotes that v is an <u>ancestor</u> of w, w is a <u>descendant</u> of v, and that v is contained in the unique tree path from r to w. Furthermore, if $v \neq w$, then v is a <u>proper</u> <u>ancestor</u> of w and w is a <u>proper</u> <u>descendant</u> of v. If $v \xrightarrow{*} w$ and (v,w) is an edge of T, v is the <u>parent</u> of w and w is the <u>child</u> of v, denoted by v --> w.

A <u>depth</u>-<u>first</u> <u>search</u> of an undirected graph G imposes a direction on the edges of G depending on the direction in which the edges are traversed. The search also partitions the edges of G into two groups: a set of <u>tree</u> <u>edges</u>, T, defining a <u>depth</u>-<u>first</u> <u>spanning</u> <u>tree</u> of G, and a set of <u>back</u> <u>edges</u>, B, which satisfy $v \xrightarrow{*} w$ or $w \xrightarrow{*} v$ in the spanning tree. Back edges are denoted by v -- w. A <u>depth</u>-<u>first</u> <u>spanning</u> <u>forest</u> is a collection of trees rooted at the vertex at which the depth-first search of each tree is begun. If G is connected, the forest is actually a tree.

An <u>adjacency</u> <u>list</u> for a vertex v is a list of all vertices w adjacent to v. Thus a graph can be represented by n adjacency lists, one for each vertex.

3. THE ST-NUMBERING

Three steps must be performed to compute an st-numbering. Each step is described by a particular algorithm which is presented in this section. The algorithms BICONNECT, PATHFINDER, and STNUMBER appear in pseudo Algol.

3.1 Step 1 : BICONNECT

Graph algorithms need a systematic method of exploring a graph. Depth-first search is a valuable technique for visiting vertices of an undirected graph. We start at some vertex v of G and select any edge (v,w) incident upon v. Traversing the edge leads us to a new vertex w. We continue the search by selecting and traversing unexplored edges incident upon the most recently reached vertex which still has unexplored edges. If depth-first search is applied to a connected graph G, each edge will be traversed exactly once. If the graph is not connected, a connected component will be searched. A new vertex is then chosen as a starting point for continuing the search.

Figure 3.1 illustrates the application of depth-first search to a graph G. The subgraph (V,T) in Fig 3.1(b) is the depth-first spanning tree generated by the search. A tree will be drawn with its root at the bottom and with the children of each vertex drawn from left to right in the order in which their edges were added to the set T. Tree edges are drawn as solid lines. These are edges which lead

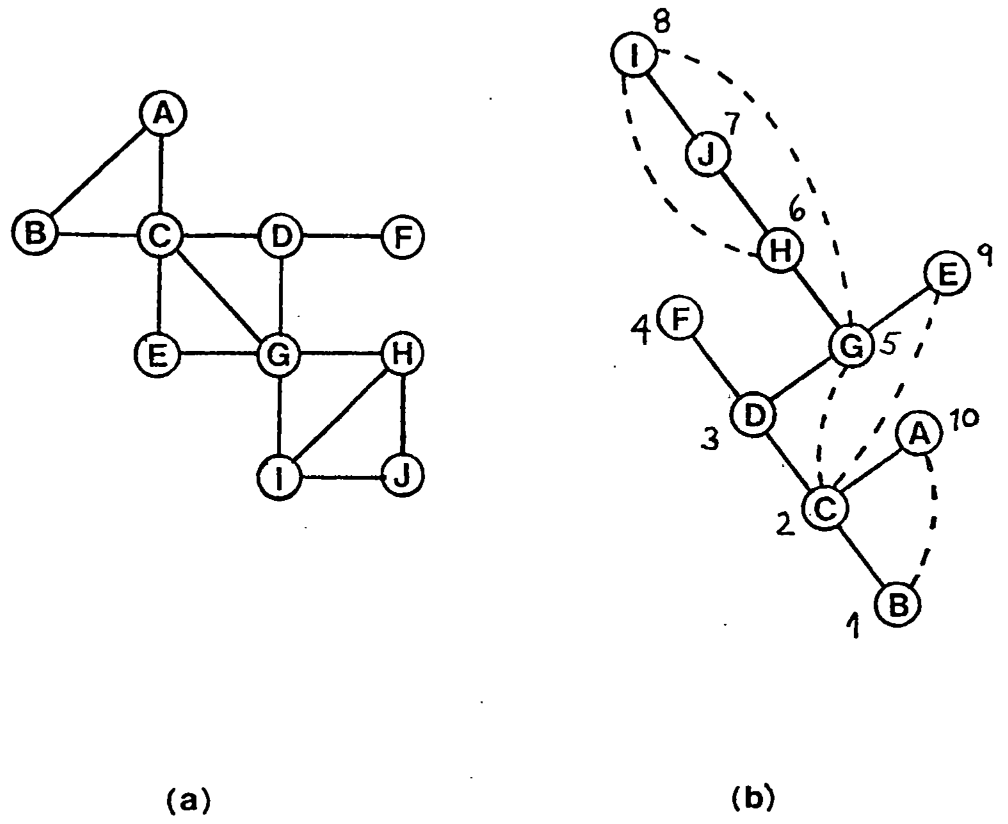


Fig. 3.1 (a) A graph and (b) its depth-first spanning tree.

to a new vertex when traversed during the depth-first search. We have included back edges in the diagram. These are edges in G, but not in T, which connect ancestors to descendants in the tree. They are represented as dashed lines. Thus for a back edge (v,w), w is an ancestor of v or v is an ancestor of w.

As well as identifying tree and back edges, the search labels the vertices in the order they are first visited. We will treat these labels as names for the vertices. For example, we can say v < w where v is an ancestor of w in the depth-first spanning tree.

The depth-first spanning tree for G is not unique. It represents only one possible search of the graph from a starting vertex s. Despite the number of spanning trees for one graph, applying the planarity test to each tree will always produce the same result.

We now look at how the BICONNECT algorithm applies a depth-first search along the edges of a graph G to divide G into its biconnected components. During the search, DFNUM and LOWPT values are assigned to the vertices. We will refer to vertices by their DFNUM values. The LOWPT value for a vertex v is the minimum of the following three values: 1) v itself, 2) u, where $u = \min\{\mathrm{LOWPT}(x) \mid x \text{ a child of } v\}$, 3) y, where $y = \min\{w \mid (v,w) \text{ a back edge}\}$. The following lemma provides a basis for finding biconnected components.

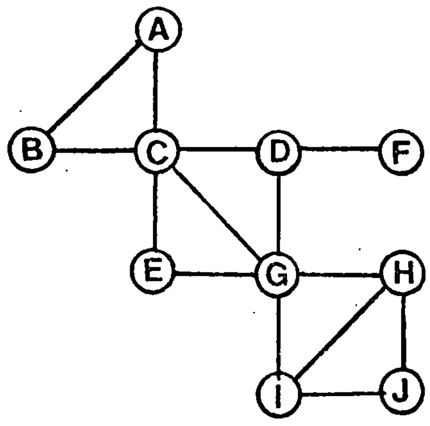
LEMMA 2 (Tarjan). If G is biconnected and $v \dashrightarrow w$ in the depth-first search spanning tree, then a) if v is not the root, $\mathrm{LOWPT}(w) < v$ and b) if v is the root, $\mathrm{LOWPT}(w) = v = 1$ and v has only one child.

PROOF. Suppose $\mathrm{LOWPT}(w) \geq v$ in part a. This implies that there is no back edge between any descendant of w, including w itself, and a proper ancestor of v. Let u be the parent of v and x a descendant of w. Any back edge from x goes to an ancestor of x. Thus it goes either to v or to a descendant of w, including w. It can never go to a proper ancestor of v according to our hypothesis. Hence every path from w to u contains v, making v an articulation point. Since G is biconnected and has no articulation points, we have a contradiction. We conclude that LOWPT(w) < v.

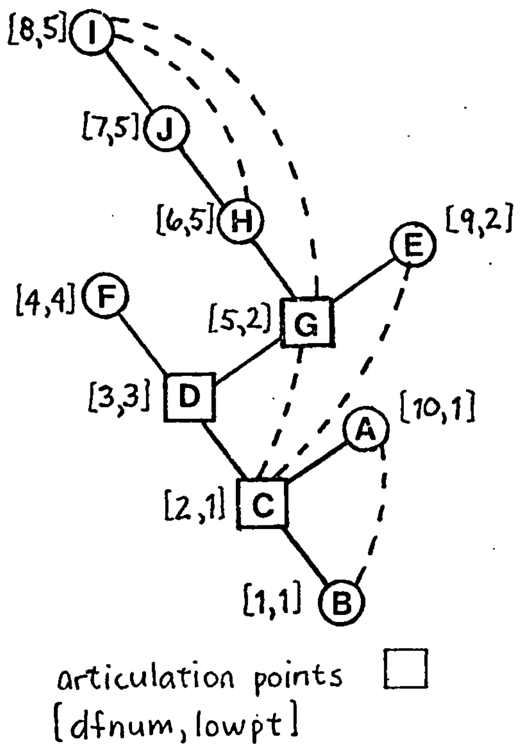
For part b, suppose LOWPT(w) $\neq$ v. Then either LOWPT(w) < v or LOWPT(w) > v. We have already shown that when LOWPT(w) > v, v is an articulation point. This is a contradiction since G is biconnected and has no articulation points. Suppose LOWPT(w) < v. It is clearly seen that if v is the root of the depth-first search spanning tree, its LOWPT value is 1. Thus the LOWPT(w) can never be less than 1 since 1 is the minimum value. This is a contradiction, hence LOWPT(w) = v = 1. The root has only one child because having more than one child means every path from one child to another contains v. Thus v is an articulation point. This is a contradiction since G is biconnected and has no articulation points.

Once LOWPT values are found for each vertex, articulation points and biconnected components can be determined during one search of a connected component.

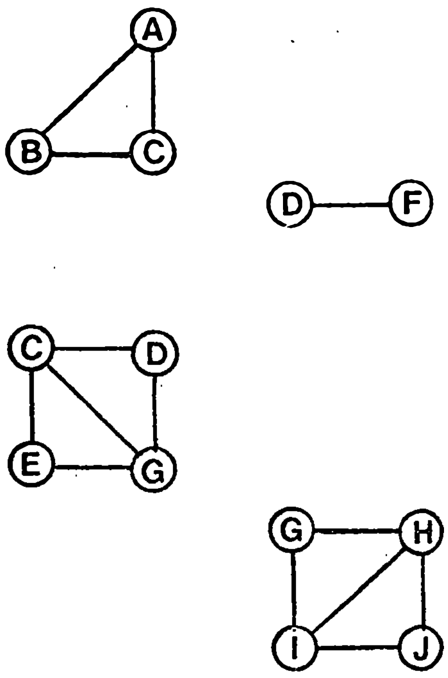
The tree of Fig 3.1(b) is reproduced as Fig 3.2(b) with indicated LOWPT values and articulation points. The biconnected components are shown in Fig 3.2(c). A list is also given of the order in which the edges were traversed in (d).



(a)



(b)



(c)

B,C	I,H
C,D	I,G
D,F	G,C
D,G	G,E
G,H	E,C
H,J	C,A
J,I	A,B

(d)

Fig. 3.2 Depth-first search of a graph: (a) graph; (b) spanning tree with LOWPT values; (c) biconnected components; (d) edge traversal.

The BICONNECT algorithm finds the biconnected components of a graph. The pseudo code is given below. Before entering BICONNECT, all vertices are marked as "new" and counters are properly initialized. An arbitrary vertex v is selected and a call, BICONNECT(v), is made. Vertex v is marked "old" and a record is kept of its depth-first search number and low point value. As edges are traversed for the first time, they are placed on a stack. This occurs at line f of BICONNECT. If an edge leads us to a vertex marked as "new," we call BICONNECT of that vertex. At some point in the program, suppose we call BICONNECT(p) and find that each of the vertices adjacent to p is now "old." We then fall back one level in the recursion and return to line j. If this tests fails, an articulation point has been found. The edges down to and including (v,w) are popped from the stack. They form a biconnected component.

When the stack is empty, a complete search of a connected component has been made. If G is connected, the process ends. Otherwise a new node is selected and the BICONNECT algorithm is repeated.

```
        procedure BICONNECT(v);
        begin
a           mark v "old";
b           DFNUM(v) <-- COUNT;
c           COUNT <-- COUNT + 1;
d           LOWPT(v) <-- DFNUM(v);
e           for each w on ADJLIST(v) do
            begin
f               add (v,w) to stack of edges if traversed for
                   the first time;
g               if w is "new" then
                begin
h                   add (v,w) to list of tree edges T;
i                   BICONNECT(w);
j                   if LOWPT(w) < v then
k                       LOWPT(v) <-- MIN (LOWPT(v), LOWPT(w))
                    else
l                       pop stack down to and including (v,w)
                end
                else
m                   if w is not the parent of v then
                    begin
n                       add (v,w) to list of back edges B;
o                       LOWPT(v) <-- MIN (LOWPT(v), DFNUM(w))
                    end
            end
        end
```

3.2 Step 2 : PATHFINDER

We assume at this point that G is a biconnected graph which has been explored using depth-first search. We have acquired information about the nodes and have generated the tree T.

This algorithm is used to partition a graph into simple paths such that the paths exhaust the edges of the graph. Before the initial call to the routine, vertices s, t, and the edge (s,t) are marked "old." All other vertices and edges in the graph G are marked "new."

Each call to PATHFINDER(v), with v as "old," produces a simple path of "new" edges before the call. The call connects the starting point v with some vertex w which was "old" before the call. Thus the initial call, PATHFINDER(s), returns a simple path from s to t not containing (s,t). The vertices and edges along the returned path are then marked as "old." If no "new" edges can be found when searching for a simple path, the procedure returns the null path.

```
        procedure PATHFINDER(v);
        begin
a           if there is a "new" back edge (v,w) with w -*-> v then
            begin
                let path be (v,w);
                mark edge "old"
            end
b           else  if there is a "new" tree edge v --> w then
                      begin
                          initialize path as (v,w);
                          while w is "new" do
                          begin
                              if there is a "new" back edge with
                                x = LOWPT(w) then
                                  add (w,x) to path
                              else
                              begin
                                  find a "new" tree edge (w,x) with
                                    LOWPT(x) = LOWPT(w);
                                  add (w,x) to path
                              end;
                              mark w "old";
                              mark edges added to path "old"
                          end
                      end
```

```
c               else  if there is a "new" back edge (v,w)
                        with v $\xrightarrow{*}$ w then
                        begin
                            initialize path as (v,w);
                            while w is "new" do
                            begin
                                find the "new" tree edge (w,x)
                                  with x --> w;
                                add to path;
                                mark w "old";
                                mark edge as "old";
                                w <-- x
                            end
                        end
                        else
d                           let path be the null path
    end
```

3.3 Step 3 : STNUMBER

The last step in computing the s-t numbering is the actual numbering of the vertices. As well as using information provided by BICONNECT, the procedure utilizes the sequence of vertices returned by the call to PATHFINDER.

The STNUMBER procedure keeps a stack of "old" vertices. The "old" nodes are those that were visited by PATHFINDER.

Initially the stack contains s on top of t. The top vertex v on the stack is deleted and PATHFINDER(v) is called. If the path returned by PATHFINDER is (v_1,v_2), (v_2,v_3) ,..., (v_{n-1},v_n), then the nodes v_{n-1}, v_{n-2} ,..., v_2, v_1 are pushed onto the stack, in that order.

If PATHFINDER(v) returns the null path, v is assigned the next available s-t number and not put back on the stack. The process is repeated until the stack is empty. At that time, all vertices of G have been s-t numbered.

```
       procedure STNUMBER;
       begin
a          mark s, t, and (s,t) "old";
b          mark all other edges and vertices "new";
c          push s on top of t in stack;
d          i <-- 0;
e          while stack is not empty do
           begin
f              let v be top vertex on stack;
g              pop v from stack;
h              PATHFINDER(v) and let path
                 be $(v_1,v_2)$ ,..., $(v_{n-1},v_n)$;
i              if path is not null then
j                  push $v_{n-1}$ ,..., $v_1$ onto the stack
```

```
                else
                    begin
k                       i <-- i + 1;
l                       stnumber(v) <-- i
                    end
            end
        end
```

3.4 Finale

Performing the above three steps results in finding the biconnected components of a graph and generating an st-numbering for each. This alone obviously does not determine planarity, but rather, is a preliminary step toward finding the answer. If each biconnected component of G can be shown to be planar, then we know from graph theory that the graph is planar. The Booth and Lueker PQ-tree algorithm does just this. The algorithm tests graphs in linear time given that the graphs are biconnected and st-numbered. Proofs of the linearity of the steps presented here can be found in the next section. A discussion of the linearity of Young's implementation of the PQ-tree algorithm can be found in [7]. A full description of the implementation is also presented in [7].

4. CORRECTNESS and LINEARITY

We provide proofs of correctness for BICONNECT, PATHFINDER, and STNUMBER. Time bounds are also presented for the algorithms.

THEOREM 1. The BICONNECT algorithm correctly finds the biconnected components of a graph G.

PROOF. Part of the BICONNECT algorithm contains the algorithm for finding connected components. Since the connectivity algorithm is well-known and correct, we will prove only that the biconnectivity algorithm works correctly on connected graphs G.

We want to prove that if the test LOWPT(w) $<$ v fails on line j, then all the edges above and including (v,w) on the stack are exactly those edges that form a biconnected component. The proof is by induction on the number of biconnected components, b, of G. The basis b = 1 implies G is biconnected. By LEMMA 2, we see that LOWPT(w) $<$ v fails when v is the root. Though the root is not an articulation point, it can be treated as one in this case. BICONNECT(w) is completed and all edges of G are on the stack. Clearly this is the correct output since the entire graph is a single biconnected component. Thus the algorithm works correctly in this case.

Suppose the induction hypothesis is true for all graphs with b biconnected components. Let G be a graph with b+1 biconnected components. Suppose LOWPT(y) < x is the first time that the test fails. BICONNECT(y) has been completed and no edges have been removed from the stack. Thus the edges above (x,y) are exactly those edges incident upon descendants of y. These are precisely the edges that make up the biconnected component containing (x,y). We are never short an edge because BICONNECT(y) has been completed. We do not store an extra edge because an additional edge would arise from a descendant of y to a proper ancestor of x, but this would alter LOWPT(y) such that LOWPT(y) < x would not fail. Thus the first biconnected component is successfully found.

Let G' be the graph that is obtained from G by deleting those edges that form this first biconnected component. After the removal of edges from the stack, the algorithm behaves exactly as it would on the graph G' except for a (trival) compression in the numbering of vertices. G' now has b biconnected components and the induction follows.

THEOREM 2. The BICONNECT algorithm requires O(n+e) steps if the graph has n vertices and e edges.

PROOF. The time required to initially mark all vertices "new" is O(n). The amount of time spent in BICONNECT(w), not counting recursive calls, is proportional to the number of vertices adjacent to w. Notice that BICONNECT(w) is called only once for a given w. During the algorithm, each edge is placed on the stack once and removed once. These operations and the calculation of LOWPT values require time proportional to e. The search for "new" start vertices upon completion of the searches of connected components takes O(n) steps. Thus BICONNECT requires time linear in n and e.

LEMMA 3. The PATHFINDER algorithm correctly finds a simple path of "new" edges from an "old" vertex y to some other "old" vertex z such that after each call to PATHFINDER(y), 1) all "old" vertices have all their ancestors "old" and 2) all tree edges on the returned path are marked "old." If there is no "new" edge (y,x) before the call to PATHFINDER(y), a null path is returned.

PROOF. Before the call to PATHFINDER, the root v, its child w, and the tree edge (v,w) are marked "old." Each call to PATHFINDER executes one of the statements a through d.

When the first call to PATHFINDER(w) is made, statement a is not chosen because there is no back edge $v \overset{*}{-}> w$. Hence choice b is examined. If the biconnected graph G does not consist of only two vertices and an edge, statement b will be chosen. This choice traverses a path (w_1,w_2), (w_2,w_3) ,..., (w_{n-1},w_n) where $w_i \overset{*}{-}> w_{i+1}$ for $1 \leq i < n-1$. Edge (w_{n-1},w_n) is a back edge where w_n --> w_{n-1} and w_n = LOWPT$(w_2) < w_1$. All vertices and edges on the path are marked "old." Thus all "old" vertices have their ancestors marked "old" and the tree edges on the path are "old." The path is simple and hence the algorithm is correct for this choice.

Choice c is selected when all nodes x, where $w \rightarrow x$, are "old." It traverses a path (w_1,w_2), (w_2,w_3) ,..., (w_{n-1},w_n) where $w_{i+1} \rightarrow w_i$ for $2 \leq i < n$ in the spanning tree. Edge (w_1,w_2) is a back edge such that $w_1 \xrightarrow{*} w_2$. Node w_n is some descendant of w_1, but not w_1 itself. All vertices and edges on this simple path are marked "old." Thus the tree edges are "old" on the path and "old" vertices have all their ancestors "old." The ancestors are "old" from the current call to PATHFINDER and earlier calls during which choice b or c was selected. Hence the algorithm performs correctly for choice c.

In statement a, the path traverses a back edge from some "old" node y to some other "old" node x such that $x \xrightarrow{*} y$ in the spanning tree. The edge (x,y) is marked "old." This choice works correctly since tree edges and ancestors of "old" nodes are "old" from previous executions of statements b and c.

Finally, if there is no "new" edge (w,x), choices a, b, and c fail. The null path is chosen and the algorithm obviously works correctly in this last case. Hence PATHFINDER performs as stated in the lemma.

LEMMA 4. The running time for PATHFINDER is O(n+e) for a graph with n vertices and e edges.

PROOF. The time spent in each call to PATHFINDER depends on the number of edges that are in the path. So after one call, the running time is O(1 + length of the path). Since PATHFINDER is called once for each vertex and each edge of the graph is in a path only once, the total time for all calls is O(n + e).

LEMMA 5. The STNUMBER algorithm correctly computes an s-t numbering of a biconnected graph G.

PROOF. Since G is biconnected, each vertex is reachable from a vertex s by a path not containing a vertex t. Each call to PATHFINDER returns a simple path. The call also ends at some "old" vertex which already occurred in some other path. Thus the last point vn in the path is not placed on the stack. No vertex is ever on the stack more than once at any one time. When PATHFINDER returns a null path, the top point on the stack is deleted and numbered. It follows that all the vertices in G are placed on stack, deleted, and numbered before t is deleted. The first deleted point, s, receives the number 1 and t, the last point, receives the number n. Any time a vertex $x \neq s$ or t is added to the stack, it is placed on top of an adjacent vertex y and has another adjacent vertex w placed on top of it. Thus $STNUM(w) < STNUM(x) < STNUM(y)$ is satisfied for any $x \neq s$ or t in the stack.

LEMMA 6. The STNUMBER algorithm requires O(n+e) steps for a graph with n vertices and e edges.

PROOF. The total amount of time to delete the vertices and assign an STNUM is O(n). The rest of the time in the algorithm is spent in PATHFINDER, which requires O(n+e) steps. Thus STNUMBER also has a time bound linear in n and e.

5. IMPLEMENTATION

This section describes some changes that have been made to Young's original implementation of the Booth and Lueker algorithm. Following this is a description of the data structures, global variables, and some local variables. Finally, input and output formats are discussed. We do not discuss Young's implementation, but use it as a "black box."

5.1 Modifications

The program for determining graph planarity incorporates the st-numbering routine and Young's implementation of the algorithm of Booth and Lueker. Pascal was used as the programming language. The entire program of Young is utilized in the planarity program as one major procedure, procedure PLTEST. Thus all the procedures which comprised the original implementation are presently local to PLTEST. All other procedures in the planarity program determine the st-numbering. In particular, procedure STNUMBER calls PLTEST after a biconnected component has been found and st-numbered. It is the job of PLTEST to determine whether the special-numbered, biconnected component is planar or is not planar.

Other changes have been made to Young's program. Four output procedures have been deleted: REPRODUCEINPUT, PRINTCHILDREN, PRINTSTRUCTURE, and PRINTSET. The input procedure, READINPUT, has been replaced with a similar procedure called FORMAT.

READINPUT required the input graphs to be biconnected and their adjacency lists to be directed from the lower numbered vertex to the higher. The vertices were assumed to be st-numbered. Instead, FORMAT builds the adjacency lists in the above manner. The vertices are already st-numbered from a previous procedure. While the adjacency lists are built, or read in as in READINPUT, the vertices are placed in either of two lists, ADJLIST or THESET. We have chosen to initialize these lists in the main body of PLTEST instead of in FORMAT.

The type declaration statements in Young's program do not appear in PLTEST. Rather, they appear at the beginning of the planarity program.

The local variable M in PLTEST has been deleted. Its occurrence in function PLANAR has been replaced by the global variable STCOUNT. STFIRST is the only argument in the parameter list for PLTEST. Its value is assigned in procedure STNUMBER. A variable J has been added to the variable list for PLTEST. It subscripts the two lists ADJLIST and THESET.

Finally, a value of 1 or 0 is assigned to the global variable RESULT in PLTEST. During the final output of results, this value is interpreted in WRITEOUT as either "is planar" or "is not planar."

5.2 Data Structures

Each node in the graph is represented by a record called VERTEX. The following fields, which are described below, make up this structure.

ID. The number which identifies the node.

DFNUM. The number assigned to a node as its position in the order of inspection during the depth-first search.

LOWPT. This number is assigned to a vertex during the depth-first search. It is the minimum of the following three values: 1) DFNUM(v), 2) u, where u = min{LOWPT(x) | x a child of v}, 3) y, where y = min{w | (v,w) a back edge}. LOWPT is used to identify articulation points.

STNEXT. This indicates which vertex is next in the st-numbering order.

ADJLIST. A pointer to the adjacency list for the node.

FLAG. An indication of whether the vertex is "available," "used," or "old." All vertices are "available" at the start. If the node has been visited during the depth-first search, it is labeled as "used." The node becomes "old" when it is placed on a path in procedure PATHFINDER. In procedure FORMAT, the nodes are marked "used" again. This allows the node to be used in another biconnected component.

The structure GRAPH is an array of records of type VERTEX.

Another record, ADJPOINT, represents an edge of the graph. Put together, these records form the adjacency lists for the graph. The following records are common to this record.

NODE. This is an array of two locations. The locations hold the vertices which make up an edge of the graph. During procedure BICONNECT, the nodes are ordered so that location one contains the ancestor/parent and location two contains the descendant/child.

NEXT. This is an array of two pointers. The pointer locations are in one-to-one correspondence with the node locations. NEXT(i) points to the location of the next node on the adjacency for NODE(i).

EDGETYPE. An indication of whether the edge is a "tree" edge, "back" edge, or "neither". Initially, all edges are "neither." The procedure BICONNECT determines edge types.

MARK. This indicates whether the edge has been placed in a path. If it has, it is marked "old." Otherwise, this field is "new."

ELINK. This is a pointer to another edge. It links the edges together as they are traversed in the BICONNECT routine. This is used instead of pushing edges onto a stack.

Three record types comprise the data constructs for procedure PLTEST. They are PQNODE, DNODE, and LINKER.

PQNODE is the most complex of the three structures. It handles three types of tree nodes: PNODE, QNODE, and LEAF. The following record fields are common to PQNODE.

NODETYPE. An indication of whether the node is a PNODE, QNODE, or a LEAF.

PARENT. This points to the immediate ancestor of the node. The field is never nil for children of P-nodes and endmost children of Q-nodes. The field is set to nil for the interior children of a Q-node. They can find their parent through their endmost siblings.

BROTHER, SISTER. These point to immediate siblings of the node. The pointers are not associated with a left or right ordering.

GROUP. Points to a sequence of full children in a Q-node. The pointer of the end node of this sequence points to the node at the other end of the sequence and vice versa. Full nodes in the interior of the sequence have this field set to nil. The nil field is also set for all other nonfull children.

LISTPLACE. This points to a node's position in the partial list.

PARTIAL. This Boolean flag indicates whether a node is partial. It is set to true when a node is placed on the queue during the partial phase. The field is never used for a LEAF node.

As mentioned above, a PQNODE can be either a PNODE, QNODE, or a LEAF. Each of these types has some additional fields. Three fields are unique to a PNODE.

EMPTYCHILDREN. This points to a child which is currently known to be empty in a marked tree. In an unmarked tree, it points to the children of the node.

FULLCHILDREN. This points to a child which is currently known to be full in a marked tree. It is set to nil in an unmarked tree.

FIRSTPARTIAL, SECONDPARTIAL. These point to the first and second children known to be partial in a marked tree. The pointers are nil in an unmarked tree.

The following two fields can be found in a Q-node.

ENDSON1, ENDSON2. These point to the two endmost children of a Q-node.

Finally, LEAF contains one unique field.

INDEX. Each LEAF is mapped to an element in the set under investigation. LEAF is an integer which is the index of the element.

In contrast to PQNODE, the structure DNODE is quite simple. DNODE represents a directed node which points to the ends of a chain of siblings. The chain is characterized by all full nodes at one end and all empty nodes at the other. The two fields of this record follow.

FULLEND. This points to the endmost full node.

EMPTYEND. This points to the endmost empty node.

The last structure to be discussed is LINKER. It is used to link various lists and queues. Its three fields follow.

FLINK. This points to forward LINKER records.

BLINK. This points to back LINKER records. It is used for insertions and deletions.

NODE. This points to a PQNODE.

5.3 Global Variables

A list of global variables used by the planarity program is explained. Following this is another list of variables significant to PLTEST. Though they are local to

PLTEST, they are global to the many nested procedures in PLTEST.

MAX. This represents the maximum number of vertices in the input graph. It must be changed to allow graphs with more than 30 vertices to be tested.

MAXPLUS1. This is a constant with a value of one more than MAX.

GPH. A reference to the array of vertices GRAPH.

APTR. A pointer to ADJPOINT. It is used for scanning adjacency lists of vertices.

TOP. This always points to the top of the list which stores edges as they are traversed during the depth-first search.

TOTAL. The number of vertices in the graph.

SINGLEPTS. The number of vertices with no incident edges.

COUNT. A counter incremented with each call to BICONNECT. Its value is used as the DFNUM of a vertex.

COMP. The number of the component under investigation.

BCOMP. The number of the biconnected component under investigation.

STN. A value incremented with each node popped from the stack in STNUMBER. Its value is used as the STNUM of a node.

STAK. A reference to the vector STACK. Both STNUMBER and PATHFINDER use STAK for storing vertices.

DFLIST. A reference to the vector DFARRAY. The DFNUM of a node is the index of the vector. DFLIST is used for printing the table of nodes in depth-first search order.

SUBROOT. A pointer to the root of the pertinent subtree. If the subroot is a Q-node, SUBROOT points to the pseudoroot. A tree is rejected if more than one node can be a subroot.

PSEUDOROOT. A pointer to the endmost pertinent children.

POTENTIALROOT. Pointer to a node that may become the SUBROOT at a later time or that has SUBROOT as one of its descendants.

BLOCKEDCOUNT. The number of times that upward movement in the tree is blocked.

BOTHENDCOUNT. The number of times a Q-node has both end children marked as full while at least one of its interior children is not a full node. The value is decremented when all interior children are full and the Q-node becomes a full node.

ADJLIST. The set of adjacency lists for the graph. The ith list contains the leaves representing the edges from vertex i to a higher numbered vertex.

THESET. The set of lists in which the ith list contains the leaves representing the edges from vertex i to a lower numbered vertex.

5.4 Input and Output

The input for the planarity program is any graph except the null graph and the graph with n vertices, n $\geq$ 1, and 0 edges. These graphs are obviously nonplanar. The graph is represented as a set of adjacency lists. In this implementation, vertex labels are expected to be integer.

The first datum read is TOTAL, the number of vertices in the graph. The vertex number and its adjacency list, enclosed by parentheses, follow. The adjacency list is written so that vertex i is less than all members in its adjacency list. Thus for a connected graph with n vertices and e edges, the nth vertex and its adjacency list can be eliminated because no adjacent vertices are greater than n. Nodes with no incident edges are also absent. The first integer of the list immediately follows the left parenthesis and the last integer is immediately followed by the right parenthesis. The end of input for a graph is designated by a period immediately after the right parenthesis of the final adjacency list. Since the program can handle more than one graph as input, the period can be replaced by a semicolon which separates the graphs. A graph and its input format are shown in Fig. 5.1.

```
8
1(2 3 4 5 6)
2(3 6)
3(4)
4(5)
5(6).
```

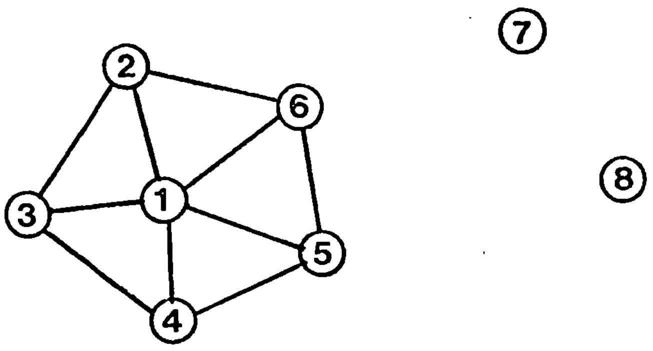
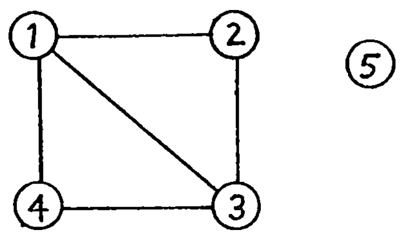


Fig. 5.1 Input format for a graph.

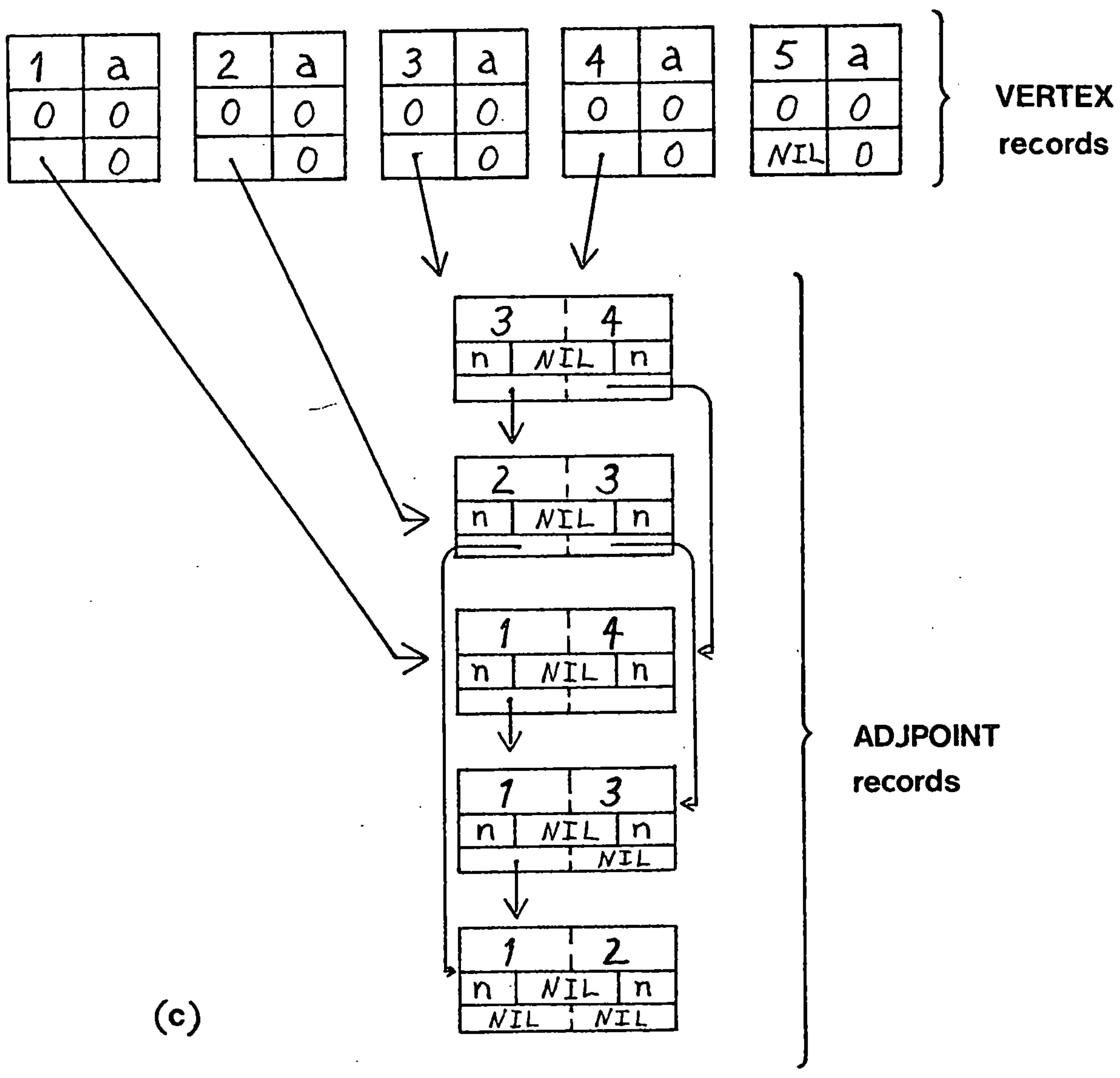
Note that the vertices are written in increasing order both across and down. This is not required, but makes things easier to read for humans. The first read statement occurs in the main body of the planarity program. TOTAL is assigned a value. Procedure CREATEVERTICES assigns the ID field of GPH[i] to the value i, where $1 \leq i \leq$ TOTAL. The procedure also initializes the other fields appropriately. The procedures involved in building adjacency lists are READANDBUILD and MAKELINK. As the former reads the input, it passes the values of the vertex v under examination and one of its adjacent nodes w to MAKELINK. This procedure allocates a record to represent the edge (v,w). The field NODE[1] is assigned v and NODE[2] is assigned w. NEXT[1] and NEXT[2] are assigned the values of the ADJLIST fields of GPH[v] and GPH[w], respectively. Then GPH[v] and GPH[w] record the current address of the record representing (v,w). Records are allocated and pointers are repositioned until the reading is complete. Fig. 5.2 shows the data structure at this stage for a sample graph. After building the data structure for the graph, the program can start its search for biconnected components.

All output from the program is handled by procedure WRITEOUT. At the start of each search of a connected component, a call to WRITEOUT displays the heading, "Component x," where x is the number of the component. When a biconnected component has been found, its vertices st-numbered, and the component tested for planarity, another call to WRITEOUT is made. Another heading, including the results of the planarity test, is printed. The vertices in the biconnected component are listed in depth-first search order below this heading. The LOWPT value and STNUM for each node is also listed. When each biconnected component is found, the heading and vertex list is printed. At the end of the search of the entire graph, the number of nodes with no adjacent vertices is printed. Program results for the graphs of Figs. 3.1 and 5.1 are given in Appendix 1. Additional sample output can also be found in Appendix 1.



```
5
1(2 3 4)
2(3)
3(4).
```

(a) (b)



(c)

Fig. 5.2 Data structure after reading input: (a) graph; (b) input data; (c) data structure.

6. CONCLUSION

This paper has presented an implementation of the depth-first search, st-numbering, and Booth and Lueker PQ-tree algorithms which combine to provide a linear-time planarity test. The test is a speeded-up version of an algorithm published by Lempel, Even, and Cederbaum. The information provided by the BICONNECT algorithm and Lemma 2 is utilized by PATHFINDER for partitioning a biconnected graph into simple paths. STNUMBER uses the paths to generate the st-numbering. Young's implementation of the Booth and Lueker PQ-tree algorithm determines whether or not the st-numbered, biconnected component is planar.

The entire listing of the planarity program appears in Appendix 2. Appendix 1 shows some sample output. Timings of results were not possible at the time of this writing.

A Honeywell 66/60 was used to run the programs. The Honeywell Pascal differs slightly from Standard Pascal. An attempt was made to keep the program standard. Two points need mentioning. Honeywell Pascal does not recognize the program statement. The "main program" of a Pascal job is a procedure named "main." The textfiles input and output are predeclared. Also, the final end statement of the program is not followed by a period.

Since machines and implementations greatly differ, it is difficult to compare the Hopcroft and Tarjan [5] algorithm with the one presented here. Possible research would be a comparison of implementations, using the same language and computer facilities, to determine storage requirements and analyze the average behavior.

REFERENCES

1. Aho A.V., J.E. Hopcroft, and J.D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, Reading, Mass., 1974.

2. Booth K.S. and G.S. Lueker, "Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms," Journal of Computer and System Sciences, Vol. 13, No. 3 (December 1976), 335-379.

3. Even S. and R.E. Tarjan, "Computing an st-numbering," Theoretical Computer Sciences, Vol. 2, (1976), 339-344.

4. Harary F., Graph Theory, Addison-Wesley, Reading, Mass., 1969.

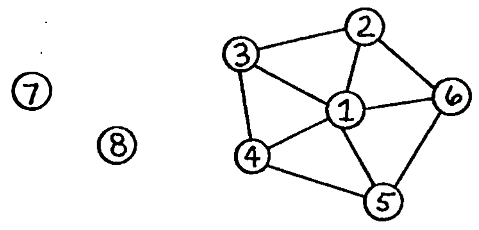
5. Hopcroft J.E. and R.E. Tarjan, "Efficient planarity testing," Journal of the ACM, Vol. 21, (1974), 549-568.

6. Lempel A., S. Even, and I. Cederbaum, "An algorithm for planarity testing of graphs," Theory of Graphs, International Symposium, Rome, July 1966, (P. Rosentiehl, Ed.), Gordon and Breach, New York, 1967, 215-232.

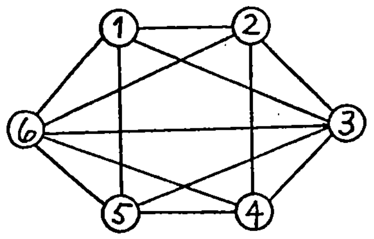
7. Young S., "Implementation of PQ-tree algorithms," Master's Thesis, Department of Computer Science, University of Washington, Seattle, Wash., 1977.

APPENDIX 1

Sample Output

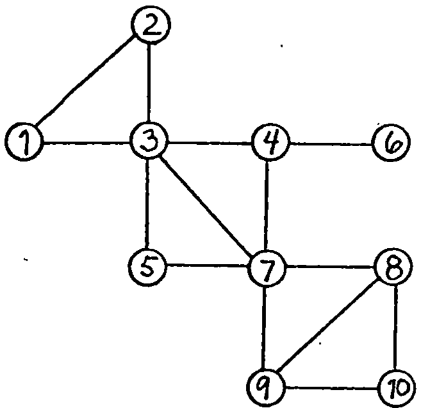


```
The graph consists of

        Component    1

            Biconnected component    1 is planar

                DFNUM     VERTEX     STNUM     LOWPT
                  1         1          6         1
                  2         6          1         1
                  3         5          5         1
                  4         4          4         1
                  5         3          3         1
                  6         2          2         1

There were    2 lonely points in the graph.
```

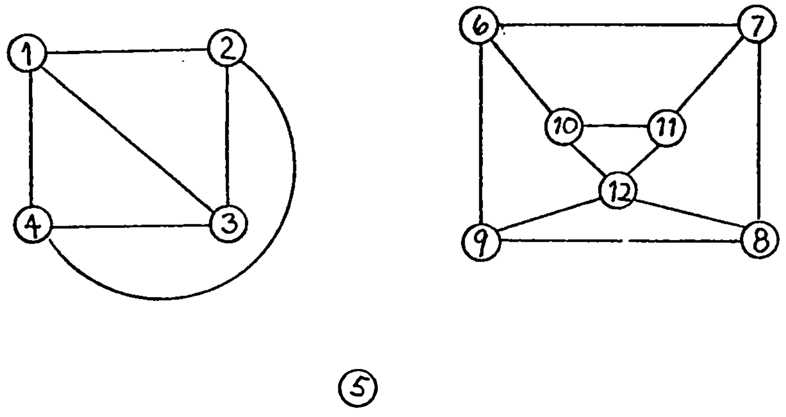


```
The graph consists of

        Component    1

            Biconnected component    1 is not planar

                DFNUM     VERTEX     STNUM     LOWPT
                  1         1          6         1
                  2         6          1         1
                  3         5          5         1
                  4         4          4         1
                  5         3          3         1
                  6         2          2         1

There were no lonely points in the graph.
```

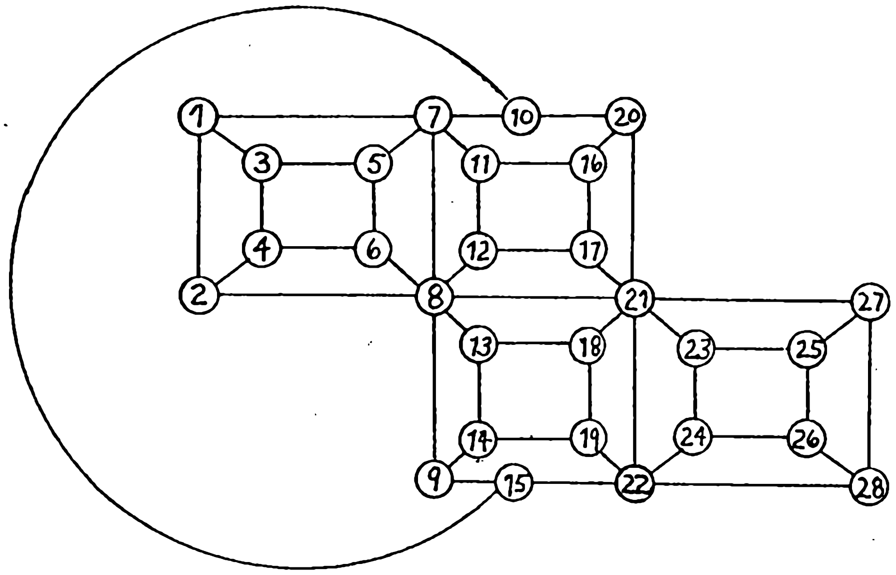
```
10
1(2 3)
2(3)
3(5 7 4)
4(7 6)
5(7)
7(9 8)
8(9 10)
9(10).
```



```
The graph consists of

      Component   1

         Biconnected component   1 is planar

            DFNUM    VERTEX    STNUM    LOWPT
              3        4         2        3
              4        6         1        4

         Biconnected component   2 is planar

            DFNUM    VERTEX    STNUM    LOWPT
              5        7         4        5
              6        8         1        5
              7       10         2        5
              8        9         3        5

         Biconnected component   3 is planar

            DFNUM    VERTEX    STNUM    LOWPT
              2        3         4        2
              3        4         1        2
              5        7         2        2
              9        5         3        2

         Biconnected component   4 is planar

            DFNUM    VERTEX    STNUM    LOWPT
              1        1         3        1
              2        3         1        1
             10        2         2        1

There were no lonely points in the graph.
```



```
The graph consists of

      Component   1

         Biconnected component   1 is planar

            DFNUM    VERTEX    STNUM    LOWPT
              1         1        4        1
              2         4        1        1
              3         3        3        1
              4         2        2        1

      Component   2

         Biconnected component   1 is planar

            DFNUM    VERTEX    STNUM    LOWPT
              1         6        7        5
              2        10        1        5
              3        12        2        5
              4        11        5        5
              5         7        6        5
              6         8        4        5
              7         9        3        5

There were   1 lonely points in the graph.
```



```
The graph consists of

      Component   1

         Biconnected component   1

            DFNUM    VERTEX    STNUM    LOWPT
              1        1        28        1
              2        7         1        1
              3       11         4        1
              4       16         7        1
              5       20         8        1
              6       21         9        1
              7       27        10        1
              8       28        11        1
              9       26        12        1
             10       25        13        1
             11       23        14        1
             12       24        15        1
             13       22        16        1
             14       19        17        1
             15       18        18        1
             16       13        19        1
             17       14        20        1
             18        9        21        1
             19       15         3        2
             20       10         2        2
             21        8        22        1
             22       12         5        3
             23       17         6        4
             24        6        25        1
             25        5        26        1
             26        3        27        1
             27        4        24        1
             28        2        23        1

There were no lonely points in the graph.
```

APPENDIX 2

Program Listing of Linear-Time Planarity Test

```
(*******************************************************************

    This program presents a linear running test for determining
    graph planarity.  It combines the implementation of the
    st-numbering algorithm and Young's implementation of the
    Booth and Lueker algorithm.  The latter is used as one
    procedure, PLTEST, in the planarity program.

    The global variables are as follows:

    max  represents the maximum number of vertices in the input
         graph.  It must be changed to allow graphs with more
         than 30 vertices to be tested.

    maxplus1  is a constant which is assigned a value of max + 1.

    gph  refers to the array of vertices GRAPH.

    aptr  points to ADJPOINT.  It is used for scanning adjacency
          lists of vertices.

    top  points to the top of the list which stores edges as
         they are traversed during the depth-first search.

    total  is the number of vertices in the input graph.

    singlepts  is the number of vertices with no adjacent nodes.

    count  is incremented with each call to BICONNECT.  Its value
           is used as the DFNUM of a vertex.

    comp  is the number of the connected component under search.

    bcomp  is the number of the biconnected component under
           investigation.

    stn  is incremented with each node popped from the stack in
         STNUMBER.  Its value is used as the STNUM of a node.
         It is also used to indicate the total number of nodes
         in a biconnected component.

    d  acts as an index variable.

    stak  is an identifier which denotes the vector STACK.  Both
          STNUMBER and PATHFINDER use STAK for storing vertices.

    dflist  is an identifier which denotes the vector DFARRAY.  The
```

```
          DFNUM of a node is the index of the vector.  DFLIST is
          used for printing the table of nodes in depth-first
          search order.

     chr  is a character variable.

     result  is the value, 0 or 1, assigned in PLTEST.  A value
          of 1 denotes the graph "is planar" and a value of
          0 says the graph "is not planar."

*****************************************************************)


procedure main;

const max = 30;
      maxplus1 = 31;

type ptr = ^adjpoint;

     vertex = record  id : integer;
                 dfnum,lowpt,stnum,stnext : integer;
                 adjlist : ptr;
                 flag : char
               end;

     adjpoint = record  node : array[1..2] of integer;
                   next : array[1..2] of ptr;
                   edgetype,mark : char;
                   elink : ptr
                 end;

     graph = array[1..max] of vertex;

     dfarray = array[1..max] of integer;

     stack = array[1..maxplus1] of integer;

     pqtype = (pnode,qnode,leaf);

     pqptr  = ^pqnode;

     positiontype = (rootoftree,childofpnode,endsonofqnode,
                  interiorchildofqnode);

     grouptype = (nogroup,endgroup,innergroup,twoendgroups,
                       twoothergroups);
     listptr =^linker;
     dptr = ^dnode;
     linker = record  node : pqptr;
                blink,flink : listptr
              end;

       node = record  parent,brother,sister,group : pqptr;
```

```
               listplace : listptr; partial : boolean;
               case nodetype : pqtype of
                  leaf : (index : integer);
                  pnode : (fullchildren,emptychildren,firstpartial,
                             secondpartial : pqptr);
                  qnode : (endson1,endson2 : pqptr)
               end;

      dnode = record  fullend,emptyend : pqptr
              end;


var gph : graph; aptr,top : ptr;
    total,singlepts,count,comp,bcomp,d,stn,result : integer;
    stak : stack; dflist : dfarray; chr : char;


procedure pltest(stfirst : integer);
(* This procedure tests a biconnected component for planarity.
   The entire procedure is a modified version of Young's
   program.  Parameter STFIRST, used by procedure FORMAT, identifies
   the vertex whose st-number is one.  Succeeding nodes to be
   st-numbered can be identified by their preceeding ones *)
var root,subroot,potentialroot,pseudoroot,p : pqptr;
    partiallist,listrear,qfront,qrear : listptr;
    blockedcount,bothendcount,n,fd,i,j : integer;
    u : listptr; theset : array[1..max] of listptr;
    reject : boolean; adjlist : array[1..max] of listptr;
    directednode : dptr; pertinentfullnode : pqptr;
    t : pqptr;

   (*****************************************************************
                          basic procedures
   *****************************************************************)

   procedure insert(p:pqptr;var head:listptr);
   (* inserts a node p into some list named head *)
   var l:listptr;
   begin
      new(l);
      if head ~= nil then head^.blink := l;
      with l^ do begin
          node := p;flink := head; blink:= nil;
         end;
      head := l;
   end;  (*insert*)

   procedure delete(l:listptr;var head:listptr);
   (* deletes a link pointed to by l from list head *)
   begin with l^ do begin
      if blink = nil
         then head := flink
         else blink^.flink :=flink;
      if flink ~= nil then flink^.blink := blink;
```

```
end; end; (* delete*)

function makenew(newtype:pqtype):pqptr;
(* creates and initializes and new pqnode of type newtype *)
var p:pqptr;
begin case newtype of
   pnode:  new(p);
   qnode:  new(p);
   leaf:   new(p)
   end;
with p^ do begin
   nodetype := newtype; brother := nil; sister := nil;
   parent := nil; group := nil; listplace := nil; partial := false;
   if nodetype = pnode
      then begin fullchildren := nil; firstpartial := nil;
         secondpartial := nil end;
  end;
makenew := p;
end;  (* makenew *)

function createuniversaltree(u:listptr):pqptr;
(* creates the universal tree with the children on the list u
   and returns the root of this tree *)
var p:pqptr;
begin
  if u = nil
     then createuniversaltree := nil
     else if u^.flink = nil
        then createuniversaltree := u^.node
        else begin
           p := makenew(pnode); p^.emptychildren := u^.node;
           createuniversaltree := p; u^.node^.parent:= p;
           u := u^.flink;
           while u ~= nil do with u^ do begin
                node^.sister := blink^.node;
                node^.parent:= p;
                blink^.node^.brother:= node;
                u := flink;
           end
        end
end;  (* createuniversaltree *)

procedure format;
(* Procedure represents the biconnected component as a set
   of adjacency lists. The vertices are directed from the
   lower numbered node to the higher numbered node.  The
   vertices are identified by their st-numbers.  As the
   vertices are read, they are placed in either of two lists:
   ADJLIST or THESET.  Local variable SS is the current
   vertex name.  N is the location (either 1 or 2) of SS in
   the array NODE.  MM is the STNUM of M.  VNUM is used as an
   index variable *)
var ss,n,m,mm,vnum : integer;  p : pqptr;
begin
```

```
   vnum := 0;
   ss := stfirst;          (* ss is the node with STNUM of 1 *)
   while gph[ss].stnext ~= 0 do
   (* while not the last st-numbered vertex *)
   begin
      vnum := vnum + 1;
      aptr := gph[ss].adjlist;       (* get an adjacent node *)
      while aptr ~= nil do
      begin
         if aptr^.node[1] = ss then
         begin
            n := 1;                  (* ss is in location one *)
            m := aptr^.node[2]
         end
         else
         begin
            n := 2;                  (* ss is in location two *)
            m := aptr^.node[1]
         end;
         if (aptr^.mark = 'o') and
            (gph[m].stnum > gph[ss].stnum) then
         begin
            mm := gph[m].stnum;
            p := makenew(leaf);
            p^.index := mm;
            insert(p,theset[mm]);
            insert(p,adjlist[vnum])
         end;
         aptr := aptr^.next[n]                (* look at the next
                                                 adjacent point *)
      end;
      ss := gph[ss].stnext  (* get the next st-numbered node *)
   end
end;  (* format *)

procedure replacepseudoroot(p:pqptr);
var sibling1,sibling2 : pqptr;
begin with pseudoroot^ do begin
  sibling1 := endson1^.brother;
  if endson1 = endson2
     then sibling2 := endson2^.sister
     else sibling2 := endson2^.brother;
  if p ~= nil then  p^.brother := sibling1;
  if p ~= nil then  p^.sister := sibling2;
  if sibling1 = nil
     then begin
        if p ~= nil then
           p^.parent := endson1^.parent;
        case endson1^.parent^.nodetype of
          pnode:  if endson1^.parent^.emptychildren=sibling1
                     then endson1^.parent^.emptychildren := p;
          qnode:  if endson1^.parent^.endson1 = endson1
                     then endson1^.parent^.endson1 := p
                     else endson1^.parent^.endson2 := p
           end;
         end
      else if sibling1^.brother = endson1
         then sibling1^.brother := p
         else sibling1^.sister := p;
   if sibling2 = nil
      then begin
         if p ~= nil then
            p^.parent := endson2^.parent;
         case endson2^.parent^.nodetype of
           pnode: if endson2^.parent^.emptychildren=endson2
                  then endson2^.parent^.emptychildren := p;
           qnode:  if endson2^.parent^.endson1 = endson2
                      then endson2^.parent^.endson1 := p
                      else endson2^.parent^.endson2 := p
           end;
         end
      else if sibling2^.brother = endson2
         then sibling2^.brother := p
         else sibling2^.sister := p;
end end;  (* replacepseudoroot *)

(**************************************************************
        procedures used in full and partial node phase
**************************************************************)

procedure initialize(s:listptr);
(* initializes all the variables for the reduce pass *)
begin
   (* initialize *) subroot := nil; potentialroot := nil;
   bothendcount := 0; blockedcount := 0;
   partiallist := nil;
   listrear := nil;
   pseudoroot := makenew(qnode); new(directednode);
   with directednode^ do begin fullend := nil;
                               emptyend := nil; end;
   qfront := s;
   if s ~= nil
      then while s^.flink ~= nil do s := s^.flink;
   qrear := s;
end;  (* initialize*)

procedure queue (p:pqptr);
(* puts a node p on the queue *)
var l:listptr;
begin new(l); l^.node := p; l^.flink:= nil;
   if qrear ~= nil then qrear^.flink :=l
   else qfront := l;
   qrear := l;
end; (* queue *)

function nextqueuednode : pqptr;
(* returns a node from the queue *)
begin
```

```
   nextqueuednode := qfront^.node;
   qfront := qfront^.flink;
   if qfront = nil then qrear := nil;
end;  (* nextqueuednode *)

function queuelength:integer;
(* returns the length if less than 2 *)
begin
   if qfront= nil
      then queuelength := 0
   else if qfront = qrear
           then queuelength := 1
         else queuelength := 2;
end;  (* queuelength *)

procedure setupqueue;
(* sets up a queue from the partiallist and resets
   the listpointers *)
begin qfront :=partiallist; qrear := listrear;
   while partiallist ~= nil do with partiallist^.node^ do
   begin
      listplace:=nil; partial:= true;
      partiallist := partiallist^.flink;
      end;
end;  (* setupqueue *)

procedure extendgroup (p,sibling:pqptr);
(* extends the group containing sibling to include p *)
begin
   if p^.brother = sibling then
   begin
      p^.brother := p^.sister;
      p^.sister := sibling;
   end;
   if sibling^.group = sibling then
      if sibling^.brother = p then
      begin
         sibling^.brother := sibling^.sister;
         sibling^.sister := p;
      end;
   p^.group := sibling^.group;
   sibling^.group := nil;
   p^.group^.group := p;
end;  (* extendgroup *)

procedure combinegroups(p:pqptr);
(* combines the groups pointed to by p's siblings
   to include p *)
var s,b:pqptr;
begin with p^ do begin
   if sister^.group = sister then
      if sister^.brother = p then
      begin
         sister^.brother := sister^.sister;
```

```
          sister^.sister := p;
       end;
    if brother^.group = brother then
       if brother^.brother = p then
       begin
          brother^.brother := brother^.sister;
          brother^.sister := p;
       end;
    s := sister^.group; b := brother^.group;
    sister^.group := nil;  brother^.group := nil;
    s^.group := b;  b^.group := s;
end; end; (* combinegroups *)

procedure resetgroup (p:pqptr);
(* resets group pointers *)
begin with p^ do begin
   group^.group := nil; group := nil;
end;end; (* resetgroup *)

function nodeposition (p:pqptr):positiontype;
(* decides where p is located *)
begin
   if p = root then
      nodeposition := rootoftree
   else if p^.parent = nil then
           nodeposition := interiorchildofqnode
        else if p^.parent^.nodetype = pnode then
                nodeposition := childofpnode
             else nodeposition := endsonofqnode
end; (* nodeposition *)

function siblinggroup (p:pqptr; var sibling:pqptr):grouptype;
(* decides what type of group surrounds an end  node p
   and returns the sibling of p *)
begin with p^ do begin
  if brother ~= nil then sibling := brother
  else sibling := sister;
  if sibling^.group = nil
     then siblinggroup := nogroup
     else if sibling^.group^.parent ~= nil
        then siblinggroup := endgroup
        else siblinggroup := innergroup;
end end;  (* siblinggroup *)

function siblingsgroup (p:pqptr;var sibling:pqptr):grouptype;
(* for an interior node p ,decides what type of groups
   surround p and returns the pertinent sibling *)
begin with p^ do
  if brother^.group ~= nil
     then if sister^.group ~= nil
        then if (sister^.group^.parent ~= nil) and
                     (brother^.group^.parent ~= nil)
           then siblingsgroup := twoendgroups
           else siblingsgroup := twoothergroups
```

```
         else begin sibling := brother;
               if brother^.group^.parent ~= nil
                  then siblingsgroup := endgroup
                  else siblingsgroup := innergroup;
                  end
     else if sister^.group ~= nil
        then begin sibling := sister;
           if sister^.group^.parent ~= nil
              then siblingsgroup := endgroup
              else siblingsgroup := innergroup
              end
        else begin siblingsgroup := nogroup;
               if brother^.partial
                  then sibling := brother
                  else sibling :=sister;
               end;
end;   (* siblingsgroup *)

procedure putonpartiallist(p:pqptr);
(* puts a node on the partiallist *)
begin with p^ do begin
  insert(p,partiallist);
  listplace := partiallist; partial:=true;
  if partiallist^.flink=nil then listrear :=partiallist;
end end;  (*putonpartiallist*)

procedure takeoffpartiallist(p:pqptr);
(* takes a node off the partiallist *)
begin with p^ do begin
   if listrear^.node = p then listrear := listrear^.blink;
   delete(listplace,partiallist);
   listplace := nil;  partial :=false;
end end;   (* takeoffpartiallist*)

procedure resetfullnode ( p:pqptr);
(* resets a full node *)
begin with p^ do begin
   emptychildren := fullchildren;
   fullchildren := nil;
end end; (* resetfullnode *)

procedure extendsubroot(p,sibling:pqptr);
(* extends the pseudoroot to include p *)
var temp:pqptr;
begin with subroot^ do begin
  if endson1 = sibling
     then endson1 := p
     else endson2 := p;
end; end; (* extendsubroot *)

procedure removefromemptychildren(p:pqptr);
(* removes a node p from the parents list of
   emptychildren *)
begin with p^ do begin
```

```
  if brother ~= nil
     then if brother^.brother = p
         then brother^.brother := sister
         else brother^.sister := sister
     else if parent^.emptychildren = p
         then parent^.emptychildren := sister;
  if sister ~= nil
     then if sister^.brother = p
        then sister^.brother := brother
        else sister^.sister := brother
     else if parent^.emptychildren = p
         then parent^.emptychildren := brother;
  brother := nil;  sister := nil;
end end;  (* removefromemptychildren *)

procedure addtofullchildren ( p:pqptr);
(* adds a node p to its parents list of full children *)
var q:pqptr;
begin
  removefromemptychildren (p);
  with p^.parent^ do begin
     p^.brother := fullchildren;
     if fullchildren ~= nil
     then fullchildren^.sister := p;
     p^.sister := nil;  fullchildren := p;
  end;
end;  (* addtofullchildren *)

procedure createpseudoroot(p,q:pqptr);
(* points the pseudoroot to the ends of the
   pertinent group *)
begin with pseudoroot^ do begin
   endson1 := p; endson2:= q;
   subroot := pseudoroot;
end;end;   (* createpseudoroot *)

(******************************************************************
           procedure used in findpertinentsubroot
******************************************************************)

procedure findgroup(p:pqptr; var fullend,partialend:pqptr);
(* for a q node it finds where the partial son and the full
  sons are located and returns a pointer to the partialend and
  the fullend *)
begin with p^ do
   if endson1^.group ~= nil
      then fullend := endson1
      else if endson2^.group ~= nil
         then fullend := endson2
         else if endson1^.partial
            then begin fullend:=endson1;partialend:=endson1 end
            else begin fullend:=endson2;partialend:=endson2 end;
  with fullend^.group^ do
    if brother ~= nil
```

```
        then if brother^.partial
           then partialend := brother
           else if sister ~= nil
              then if sister^.partial
                 then partialend := sister
                 else partialend := fullend^.group
              else partialend := fullend^.group
        else if sister ~= nil
           then if sister^.partial
              then partialend := sister
              else partialend := fullend^.group;
 end;   (* findgroup *)

 (**************************************************************
            procedures used with process pertinent
                 subtree and makedirectednode
 **************************************************************)

 procedure find1(p:pqptr; var partialson,fullend,emptyend,
                 fullsibling,emptysibling:pqptr);
 (* used in makedirectednode to find the sibling which is full
   and the one which is empty. it also helps to decide which way
   to point the directed node *)
begin with p^ do begin
   if (endson1^.group ~= nil) or (endson1^.partial and
                (endson2^.group = nil))
      then begin fullend:=endson1;emptyend:=endson2; end
      else begin fullend:= endson2; emptyend:= endson1; end;
   if fullend^.group ~= nil
      then begin
         fullsibling := fullend^.group;
         if fullsibling^.brother^.partial
            then partialson := fullsibling^.brother
            else if fullsibling^.sister^.partial
               then partialson := fullsibling^.sister
               else partialson := nil;
          if partialson ~= nil
             then if partialson^.brother = fullsibling
                then emptysibling := partialson^.sister
                else emptysibling := partialson^.brother;
          resetgroup (fullend);
       end
    else begin
      partialson := fullend; fullsibling := nil;
      if partialson^.brother = nil
         then emptysibling := partialson^.sister
         else emptysibling := partialson^.brother;
      end;
 end; end;  (*find1 *)

 procedure find2(partialson:pqptr; var fullsibling,
                 emptysibling : pqptr);
 (* does the same for processpertinent subtree as find1 did for
    makedirectednode. see above procedure *)
```

```
begin with partialson^ do
  if brother = nil then
    begin emptysibling := brother; fullsibling := sister; end
  else  if brother^.partial or (brother^.group ~= nil) then
          begin emptysibling := sister;
                fullsibling := brother
          end
        else begin emptysibling := brother;
                   fullsibling := sister
             end;
  if fullsibling~=nil then
    if (fullsibling^.group=fullsibling) and
             (fullsibling^.brother=partialson) then
      begin
        fullsibling^.brother := fullsibling^.sister;
        fullsibling^.sister:=partialson;
      end;
end;  (* find2 *)

function makefullparent(p:pqptr):pqptr;
(* returns nil if p is nil, p if p has no siblings,otherwise
   a new parent is created for the full children *)
var q:pqptr;
begin
  if p = nil then
     makefullparent := nil
  else begin
          p^.parent^.fullchildren := nil;
          if p^.brother = nil then
             begin p^.parent := nil; makefullparent := p; end
          else begin q := makenew(pnode); q^.emptychildren := p;
                     while p~=nil do
                       begin p^.parent := q;
                             p := p^.brother
                       end;
                     makefullparent := q;
               end
       end
end;  (* makefullparent *)

function makeemptyparent (p:pqptr):pqptr;
(* returns nil if p is nil, p if p has no siblings or p's parent
   if p does have siblings *)
var q:pqptr;
begin
  if  p = nil
      then makeemptyparent := nil
      else with p^ do
        if (brother = nil)  and (sister = nil)
           then begin parent := nil; makeemptyparent:= p; end
           else begin makeemptyparent := parent;
                parent^.brother := nil; parent^.sister := nil;
                parent^.parent := nil;
                end;
```

```
end; (* makeemptyparent *)

function initialdirectednode(fullnode,emptynode:pqptr):dptr;
(* sets up the first directednode *)
begin
  with directednode^ do begin
    fullend := fullnode;
    emptyend:= emptynode;
    fullnode^.brother := emptynode;
    if emptynode ~= nil then
       emptynode^.brother := fullnode;
end; end; (* initialdirectednode *)

procedure merge(directednode:dptr; partialson,fullsibling,
                        emptysibling:pqptr);
(* merges the directed node with the full and partial sibling *)
begin with directednode^ do begin
  if fullend^.brother = nil
     then fullend^.brother := fullsibling
     else fullend^.sister := fullsibling;
  if fullsibling ~= nil
     then if fullsibling^.brother = partialson
        then fullsibling^.brother := fullend
        else fullsibling^.sister := fullend;
  if emptyend^.brother = nil
     then emptyend^.brother := emptysibling
     else emptyend^.sister := emptysibling;
  if emptysibling ~= nil
     then if emptysibling^.brother = partialson
         then emptysibling^.brother := emptyend
         else emptysibling^.sister := emptyend;
end; end; (* merge *)

procedure replace(p,q:pqptr);
(* replaces the new qnode subroot for the old pnode subroot*)
begin with p^ do begin
   if p= root then root := q;
   q^.parent := parent;
   q^.sister := sister;
   q^.brother := brother;
   if sister ~= nil then  if sister^.brother = p
      then sister^.brother := q else sister^.sister := q;
   if brother ~= nil then  if brother^.brother = p
      then brother^.brother := q else brother^.sister := q;
   if parent ~= nil then case parent^.nodetype of
     pnode: if parent^.emptychildren = p then
                parent^.emptychildren := q;
     qnode: if parent^.endson1=p then parent^.endson1 := q
                else if parent^.endson2 = p then
                        parent^.endson2 := q
   end;
end; end;  (* replace *)

procedure addtochildren(q,p:pqptr);
```

```
(* add q to p's emptychildren *)
begin with p^ do begin
  q^.parent := p; q^.brother := emptychildren;
  if emptychildren^.sister = nil
     then emptychildren^.sister := q
     else emptychildren^.brother := q;
  emptychildren := q;
end;  end;   (* addtochildren *)

procedure createfamily(q,firstpartial,fullnode,
                          secondpartial : pqptr);
(* q becomes the parent of the rest of the parameters *)
begin
   q^.endson1 := firstpartial;
   firstpartial^.parent^.firstpartial := nil;
   firstpartial^.parent := q;
   firstpartial^.brother := fullnode;
   if fullnode ~= nil
      then begin
         fullnode^.group := fullnode;
         fullnode^.sister := firstpartial;
         fullnode^.brother:= secondpartial;
         if secondpartial ~= nil
           then begin
              secondpartial^.sister := fullnode;
              q^.endson2 := secondpartial;
              secondpartial^.parent^.secondpartial := nil;
              secondpartial^.parent := q;
              end
           else begin
              q^.endson2:= fullnode; fullnode^.parent := q;
              end
         end
      else begin
         firstpartial^.brother := secondpartial;
         secondpartial^.brother := firstpartial;
         q^.endson2 := secondpartial;
         secondpartial^.parent^.secondpartial := nil;
         secondpartial^.parent := q;
         end
end;   (* createfamily *)

(*****************************************************************
                    main reduce procedures
*****************************************************************)

procedure fullnodephase;
(* starting with a queue of full leaves, the procedure moves
   up the tree finding all the full nodes *)
var groupend:pqptr; sibling:pqptr;
begin
while queuelength > 0 do begin
   p := nextqueuednode;
   with p^ do case nodeposition(p) of
   rootoftree : subroot := p;
   childofpnode: begin
                 if ~ parent^.partial
                    then putonpartiallist(parent);
                 addtofullchildren(p);
                 if parent^.emptychildren = nil
                    then begin queue (parent);
                          takeoffpartiallist(parent);
                          resetfullnode(parent);
                         end;
                 end;
   endsonofqnode: begin
                  if parent^.partial
                     then bothendcount := bothendcount +1
                     else putonpartiallist(parent);
                  case siblinggroup(p,sibling) of
                  nogroup:  group := p;
                  endgroup: begin queue(parent);
                             takeoffpartiallist(parent);
                             bothendcount:=bothendcount -1;
                             resetgroup(sibling);
                             end;
                  innergroup: begin extendgroup(p,sibling);
                             blockedcount := blockedcount -1;
                             end
                  end
                  end;
  interiorchildofqnode: case siblingsgroup(p,sibling) of
      nogroup:  begin group:= p; groupend := p;
                      blockedcount := blockedcount + 1;
                end;
      innergroup,endgroup: begin
                  extendgroup(p,sibling);
                  groupend := p;
                 end;
      twoendgroups: begin
                  queue(sister^.group^.parent);
                  bothendcount := bothendcount - 1;
                  takeoffpartiallist(sister^.group^.parent);
                  resetgroup(sister); resetgroup(brother);
                  end;
      twoothergroups: begin blockedcount := blockedcount - 1;
                     groupend := sister^.group;
                     combinegroups(p);
                    end
      end
   end;
end;
if (blockedcount > 1 )or(bothendcount > 0) then reject := true;
if (blockedcount = 1) and (partiallist = nil) then
   createpseudoroot(groupend,groupend^.group);
end; (* fullnodephase *)

procedure partialnodephase;
```

```
(* from a queue of potential partial nodes all the rest are
   found and all the reject cases are tested for *)
var sibling:pqptr;
begin
  setupqueue;
  while (blockedcount + queuelength > 1) and ~ reject do begin
    p := nextqueuednode;
    case nodeposition(p) of
     rootoftree : begin
                     blockedcount := blockedcount + 1;
                     potentialroot := p;
                  end;
     childofpnode : with p^.parent^ do
                       if firstpartial = nil then
                         begin if ~ partial then
                               begin
                                  queue(p^.parent);
                                  partial:= true
                               end;
                               removefromemptychildren(p);
                               firstpartial := p;
                         end
                       else if subroot = nil then
                            begin
                               secondpartial := p;
                               removefromemptychildren(p);
                               subroot := p^.parent;
                               subroot^.partial := false;
                            end
                            else reject := true;
     endsonofqnode: with p^ do case siblinggroup(p,sibling) of
          nogroup:  if sibling^.partial then
                       if sibling = potentialroot then
                          if subroot = nil then
                              begin potentialroot := nil;
                                createpseudoroot(p,sibling);
                              end
                          else reject := true
                        else begin
                               potentialroot := p;
                               blockedcount := blockedcount+1;
                             end
                    else if ~ parent^.partial then
                             begin queue(parent);
                                   parent^.partial := true;
                             end
                          else reject := true;
          innergroup : if subroot = nil then
                          createpseudoroot(p,sibling^.group)
                       else if (pseudoroot^.endson1 = sibling)
                          or (pseudoroot^.endson2=sibling) then
                               extendsubroot(p,sibling)
                            else reject := true;
          endgroup:   (* all okay *)
          end;
     interiorchildofqnode: with p^ do
          case siblingsgroup(p,sibling) of
            nogroup: if sibling^.partial and
                          (sibling=potentialroot) then
                        if subroot ~= nil then
                           reject := true
                        else begin
                              createpseudoroot(p,sibling);
                              potentialroot := nil;
                            end
                     else begin
                             potentialroot := p;
                             blockedcount := blockedcount+1;
                           end;
            innergroup: if subroot = nil then
                           createpseudoroot(p,sibling^.group)
                        else if (pseudoroot^.endson1=sibling) or
                                (pseudoroot^.endson2=sibling) then
                                   extendsubroot(p,sibling)
                                else reject := true;
            endgroup: (* all okay *);
            twoothergroups: reject := true
          end
    end;
    if blockedcount > 1 then reject := true;
  end; (* end of while statement *)
  if queuelength = 1 then potentialroot := nextqueuednode;
end; (* partialnodephase *)

procedure findpertinentsubroot;
(* process now moves down the tree to find the real subroot *)
var p,fullend,partialend:pqptr;
begin
  if potentialroot ~= nil then
     p := potentialroot
  else p := subroot;
  while (p ~= subroot) and ~ reject do with p^ do begin
     partial := false;
     case nodetype of
       pnode:  if fullchildren = nil then
               begin
                  addtochildren(firstpartial,p);
                  p := firstpartial;
                  p^.parent^.firstpartial := nil;
               end
               else if subroot = nil then
                       subroot := p
                    else reject := true;
       qnode:  begin
                  findgroup(p,fullend,partialend);
                  if (fullend = partialend) and
                      partialend^.partial then
                     p := partialend
                  else if subroot = nil then
                          createpseudoroot(fullend,partialend)
                       else if (subroot^.endson1 = partialend)
                         or (subroot^.endson2=partialend) then
                               p := subroot
                            else reject := true;
               end
        end
     end
  end;  (* findpertinentsubroot*)

  function makedirectednode(p:pqptr):dptr;
  (* the recursive procedures that transforms partial nodes
     into directed nodes *)
  var partialson,fullnode,emptynode,fullend,emptyend,fullsibling,
        emptysibling:pqptr;
  begin
    with p^ do begin
     partial := false;
     case nodetype of
      pnode:  begin
                fullnode := makefullparent(fullchildren);
                emptynode := makeemptyparent(emptychildren);
                if firstpartial = nil then
                begin
                   pertinentfullnode := fullnode;
                   makedirectednode :=
                       initialdirectednode(fullnode,emptynode);
                end
                else begin
                      directednode:=makedirectednode(firstpartial);
                      firstpartial := nil;
                      merge(directednode,nil,fullnode,emptynode);
                      if fullnode ~= nil then
                         directednode^.fullend := fullnode;
                      if emptynode ~= nil then
                         directednode^.emptyend := emptynode;
                     end;
              end;
      qnode:  begin
                findl(p,partialson,fullend,emptyend,fullsibling,
                                          emptysibling);
                if partialson ~= nil then
                begin
                   directednode := makedirectednode(partialson);
                   merge(directednode,partialson,fullsibling,
                                         emptysibling);
                end
                else pertinentfullnode := fullsibling;
                if fullend ~= partialson then
                begin
                  directednode^.fullend := fullend;
                  fullend^.parent := nil;
                end;
                if emptyend ~= partialson then
                begin
                  directednode^.emptyend := emptyend;
                  emptyend^.parent := nil
                end;
              end
       end;  (* of case *)
      end;  (* of with statement *)
      makedirectednode := directednode;
    end; (* makedirectednode *)

    procedure processpertinentsubtree;
    (* main reduce procedure that processes the tree *)
    var q,fullnode,partialson,fullsibling,emptysibling,dad:pqptr;
    begin if subroot~=nil then with pseudoroot^ do begin
      subroot^.partial := false;
      if subroot^.nodetype = pnode then
         if subroot^.firstpartial = nil then
            if subroot^.fullchildren = nil
               then begin
                  endson1 := subroot; endson2 := subroot;
                  end
               else begin
                  fullnode := makefullparent(subroot^.fullchildren);
                  addtochildren(fullnode,subroot);
                  endson1 := fullnode; endson2 := fullnode;
                  end
            else begin
               q := makenew(qnode);
               fullnode := makefullparent(subroot^.fullchildren);
               createfamily(q,subroot^.firstpartial,fullnode,
                     subroot^.secondpartial);
               if subroot^.emptychildren = nil
                  then replace(subroot,q)
                  else addtochildren(q,subroot);
               endson1:= q^.endson1; endson2 := q^.endson2;
               end;
      if endson1^.partial then begin
         partialson := endson1; dad := partialson^.parent;
         find2(partialson,fullsibling,emptysibling);
         directednode := makedirectednode(partialson);
         if fullsibling^.partial then
            directednode^.fullend^.group := directednode^.fullend;
         directednode^.emptyend^.parent := dad;
         merge(directednode,partialson,fullsibling,emptysibling);
         if dad ~= nil then if dad^.endson1 = partialson
                 then dad^.endson1 := directednode^.emptyend
                 else dad^.endson2 := directednode^.emptyend;
         endson1 := pertinentfullnode;
         end;
      if endson2^.partial then begin
         partialson := endson2; dad := partialson^.parent;
         find2(partialson,fullsibling,emptysibling);
         directednode := makedirectednode(partialson);
```

```
        directednode^.emptyend^.parent := dad;
        merge(directednode,partialson,fullsibling,emptysibling);
        if dad ~= nil then if dad^.endson2 = partialson
                 then dad^.endson2 := directednode^.emptyend
                 else dad^.endson1 := directednode^.emptyend;
        endson2 := pertinentfullnode;
        end;
    if endson1^.group ~= nil
       then resetgroup(endson1)
       else if endson1~=endson2 then resetgroup(fullsibling);
  end; end; (* processpertinentsubtree *)

  (**************************************************************
                        reduce and main program
  **************************************************************)

  procedure reduce(t:pqptr;s:listptr);
     begin
       initialize(s);
       fullnodephase;
       partialnodephase;
       findpertinentsubroot;
       if ~ reject then
           processpertinentsubtree
     end;  (* reduce *)

  function planar :boolean;
  begin
     reject := false;
     root := createuniversaltree(adjlist[1]);
     for i := 2 to stn-1 do
     begin
        if ~ reject then reduce(root,theset[i]);
        if ~ reject then
        begin
           t := createuniversaltree(adjlist[i]);
           replacepseudoroot(t);
        end;
     end;
     planar := ~ reject;
  end;  (* planar *)

     (**********  start of main body of pltest  **********)

begin
   for j := 1 to stn do
   begin
      theset[j] := nil;
      adjlist[j] := nil
   end;
   format;
   if planar then
      result := 1
   else result := 0
```

```
end;  (* pltest *)

(*******************************************************************
                    end of procedure pltest
*******************************************************************)


procedure createvertices;
(* Procedure assigns vertex n to location GPH[n] in the list
   of records and initializes the fields of the records.  The
   local variable I is used as a counter *)
var i : integer;
begin
   for i := 1 to total do
   begin
      with gph[i] do
      begin
         id := i;         (* set vertex name to value of index i *)
         dfnum := 0;          (* set depth-first number to zero *)
         lowpt := 0;                 (* set low value to zero *)
         stnum := 0;                 (* set st number to zero *)
         stnext := 0;            (* set next st-number to zero *)
         adjlist := nil;   (* set adjacency list pointer to nil *)
         flag := 'a';            (* mark vertex as "available" *)
      end
   end
end;  (* createvertices *)


procedure makelink (j,k : integer);
(* Procedure forms the adjacency lists for each vertex.
   Parameters J and K are vertices read as input and passed
   from READANDBUILD *)
begin
   new(aptr);          (* allocate a new record for an edge of the
                          graph and assign its address to aptr *)
   with aptr^ do
   begin
      next[1] := gph[j].adjlist;  (* next[1] points to most recent
                                     node added to adj. list of j *)
      next[2] := gph[k].adjlist;
      node[1] := j;
      node[2] := k;
      edgetype := 'n';            (* mark the edge as "neither" *)
      elink := nil;
      mark := 'n'                     (* mark the point as "new" *)
   end;
   gph[j].adjlist := aptr;     (* adjacency list pointer of vertex
                                  j points to last added vertex on
                                              its adjacency list *)

   gph[k].adjlist := aptr
end;  (* makelink *)
```

```
procedure readandbuild;
(* Procedure reads the graph so that adjacency lists can be built
   Local variable I is the vertex under examination.  NUM
   represents an adjacent point of I.  The input graph is
   represented as a set of adjacency lists.  The first number read
   is the total number of vertices in the graph.  Adjacency
   lists are written on the lines following this number.  A
   vertex number appears with a left parenthesis immediately
   following it.  All vertices adjacent to it and having numbers
   greater than it are added to the adjacency list for that vertex.
   These numbers are delimited by blanks.  The last adjacent node
   is immediately followed by a right parenthesis.  The last right
   parenthesis of the last adjacency list of a graph is immediately
   followed by a period.  This signifies the end of the data.  If
   more than one graph will be tested, the period must be
   replaced by a semicolon.  This signifies more data is to come *)
var i,num : integer;
begin
   chr := ' ';
   while not (chr in [';', '.']) do
   (* while not at end of data for a graph *)
   begin
      read (num);                    (* read the first vertex *)
      i := num;
      read(chr);
      while chr ~= ')' do
      (* while there still exist adjacent points *)
      begin
         read(num);                  (* get an adjacent point *)
         makelink(i,num);         (* add num to adj. list of i *)
         read(chr)
      end;
      read(chr)
   end
end;  (* readandbuild *)


procedure popstack (vv,ww : integer);
(* Procedure pops edge stack down to and including the edge
   (vv,ww).  These edges form a biconnected component.  Before
   leaving the procedure, the vertices vv and ww and the edge
   (vv,ww) are marked old in preparation for the st-numbering.
   The local variable FROM represents the vertex you came
   from and the variable TTO is the vertex you went to as
   determined during the depth-first search *)
var from,tto : integer;
begin
   from := top^.node[1];        (* vertices making up the top edge
                                          in the stack are assigned to
                                             variables from and tto *)
   tto := top^.node[2];
   while (from ~= vv) or (tto ~= ww) do
   (* while the edge (vv,ww) hasn't been reached *)
   begin
```

```
      if gph[from].flag = 'u' then
      (* if it is marked "used" *)
         gph[from].flag := 'n';                        (* mark it "new" *)

      if gph[tto].flag = 'u' then
         gph[tto].flag := 'n';

      top := top^.elink;           (* move pointer down one position
                                                 in the edge stack *)
      from := top^.node[1];
      tto := top^.node[2]
   end;
   gph[from].flag := 'o';                 (* mark vertex vv "old" for
                                                     st-numbering *)
   gph[tto].flag := 'o';
   top^.mark := 'o';            (* mark edge "old" for stnumbering *)
   top := top^.elink             (* move pointer down one position *)
end;  (* popstack *)


function pathtried (kind : char; i,j,pt,test : integer) : boolean;
(* This routine returns "true" if a path is found between
   two vertices and "false" otherwise.  Local variable SWITCH is
   a Boolean flag.  It is set to "true" when an edge has been found.
   Parameter KIND designates the type of edge that is required -
   "tree" or "back."  Parameter PT is the start node for the path.
   TEST is the selector for the case statement.  It further specifies
   the type of edge to be traversed.  The integers I and J
   determine the direction to take in the tree from vertex PT.
   If w is an adjacent node of PT, then

        i       j
   --------------------
        1       2         ---->    pt is the parent/ancestor of w

        2       1         ---->    pt is the son/descendant of w *)

var switch : boolean;
begin
   aptr := gph[pt].adjlist;      (* look at the first adjacent node
                                                    in the list *)
   switch := false;
   while (aptr ~= nil) and (switch = false) do
   (* while there remain adjacent nodes to investigate and
            while a path has not been found *)
     if (aptr^.node[i] = pt) then
     (* if the node is in the correct location *)
        if (aptr^.edgetype = kind) and (aptr^.mark = 'n') then
        (* if the edge is the correct kind and "new" *)

           case test of
              1:  switch := true;

              2:  if (gph[pt].lowpt =
                     gph[aptr^.node[j]].dfnum) then
                        switch := true
                  else
                     aptr := aptr^.next[i];

               3:  if (gph[pt].lowpt =
                     gph[aptr^.node[j]].lowpt) then
                        switch := true
                  else
                     aptr := aptr^.next[i]
            end  (* case *)
         else
            aptr := aptr^.next[i]                  (* look at the next
                                                      adjacent node *)
      else
         aptr := aptr^.next[j];        (* look at the next adjacent
                                    node in the other location *)
   pathtried := switch = true
end;  (* pathtried *)


procedure pathfinder (var pbot : integer; p : integer);
(* This routine calls PATHTRIED to obtain a path.  It
   places the vertices from the returned path on a stack
   so that the start vertex is the last to be pushed.
   It also marks the visited vertices and edges as "old."
   Parameter P is the start vertex for a path.  Parameter
   PBOT indicates the next available position in the stack.
   Index variable I gives the location of the node from which
   another call to PATHTRIED will be made *)
var i : integer;
begin
   pbot := maxplus1;
   if pathtried('b',2,1,p,1) then
   begin
      gph[aptr^.node[1]].flag := 'o';  (* mark the vertex "old" *)
      aptr^.mark := 'o';                  (* mark the edge "old" *)
      stak[pbot] := p;           (* place the node in the stack *)
      pbot := pbot-1;
      stak[pbot] := aptr^.node[1]             (* place last node of
                                                 path on stack *)
   end
   else
      if pathtried('t',1,2,p,1) then
      begin
         aptr^.mark := 'o';
         stak[pbot] := p;                     (* push p on stack *)
         p := aptr^.node[2];           (* assign p's child to p *)
         while(gph[p].flag = 'n') do
         (* while there are "new" vertices *)
         begin
            if pathtried('b',2,1,p,2) then  i := 1
            (* if a back edge is found then the path
               will continue from the vertex in node[i] *)
            else
               if pathtried('t',1,2,p,3) then
                  i := 2;
            gph[p].flag := 'o';
            aptr^.mark := 'o';
            pbot := pbot - 1;
            stak[pbot] := p;
            p := aptr^.node[i]    (* p is the vertex in node[i] *)
         end;
         pbot := pbot - 1;
         stak[pbot] := p
      end
      else
         if pathtried('b',1,2,p,1) then
         begin
            aptr^.mark := 'o';
            stak[pbot] := p;
            p := aptr^.node[2];
            while gph[p].flag = 'n' do
            begin
               if pathtried('t',2,1,p,1) then
               begin
                  gph[p].flag := 'o';
                  aptr^.mark := 'o';
                  pbot := pbot - 1;
                  stak[pbot] := p;
                  p := aptr^.node[1]
               end
            end;
            pbot := pbot - 1;
            stak[pbot] := p
         end
         else
            stak[pbot] := 0        (* the null path is returned *)
end;  (* pathfinder *)


procedure writeout (mssg : integer);
(* This procedure outputs results including the number
   of connected components, biconnected components, and
   number of nodes with no adjacent vertices.  It lists
   the vertices according to their depth-first search
   number.  Local variable W acts as an index variable and
   VTX is a vertex label *)
var w, vtx : integer;
begin
   case mssg of
      1 : begin
            writeln; writeln;
            writeln('The graph consists of');
            writeln
          end;

      2 : begin
```

```
            write('There were ');
            if singlepts = 0 then
               write('no')
            else write(singlepts:3);
            write(' lonely points in the graph.');
            writeln; writeln; writeln
         end;

      3 : begin
            writeln('      Component ',comp:3);
            writeln
         end;

      4 : begin
            write('         Biconnected component ',bcomp:3);
            if result = 1 then
               writeln(' is planar')
            else
               writeln(' is not planar');
            writeln;
            writeln('            DFNUM   VERTEX   STNUM   LOWPT');

            for w := 1 to total do
            (* look through the list of vertices indexed by
               depth-first number *)
               if dflist[w] ~= 0 then
               begin
                  vtx := dflist[w];
                  write('            ',w:3);
                  write('     ',vtx:3);
                  write('      ',gph[vtx].stnum:3);
                  writeln('     ',gph[vtx].lowpt:3);
                  dflist[w] := 0      (* replace the vertex label
                                      with a zero at location w *)
               end;
            writeln; writeln
         end
   end  (* case *)
end;  (* writeout *)


procedure stnumber (t,s : integer);
(* This procedure does the numbering of the vertices.  It pops
   the vertices placed on the stack from the PATHFINDER
   routine.  The last node on the path is not popped.  When
   a null path is returned by PATHFINDER, the top node on the
   st-number stack is popped and assigned a number.  This
   routine and PATHFINDER share the same stack.  PATHFINDER
   pushes vertices up into the stack and STNUMBER pushes
   vertices down onto the stack.  Thus STTOP identifies the top
   position in the stack which holds the vertices ready for the
   stnumbering.  Variable PTHBOT identifies the "top" of the stack
   which holds the vertices that were traversed during a call to
   PATHFINDER.  DFN is the depth-first search number, PREV identifies
   the node that was previously st-numbered, and FIRST gives the
   label of the node that was st-numbered first *)
var sttop,pthbot,dfn,prev,first : integer;
begin
   popstack(t,s);          (* get edges for biconnected component *)
   sttop := 1;
   stak[sttop] := t;       (* place t in lowest position in stack *)
   sttop := sttop + 1;
   stak[sttop] := s;                 (* push s on top of t in stak *)
   first := s;
   prev := s;
   stn := 0;                  (* initialize stnumber variable to 0 *)
   while sttop >= 1 do            (* while the stak is not empty *)
   begin
      s := stak[sttop];                (* s is current top element *)
      sttop := sttop-1;                                  (* pop s *)
      pathfinder(pthbot,s);             (* find a path from s to t *)
      if stak[pthbot] ~= 0 then  (* check if null path returned *)
      begin
         pthbot := pthbot + 1;         (* don't include last node on
                                                    returned path *)
         while pthbot <= maxplus1 do
         begin
            sttop := sttop + 1;
            stak[sttop] := stak[pthbot];       (* transfer vertices
                                               from path to stak *)
            pthbot := pthbot + 1
         end
      end
      else
      begin
         stn := stn + 1;
         gph[s].stnum := stn;              (* assign an st-number *)
         gph[s].flag := 'u';   (* mark the node as "used" again *)
         dfn := gph[s].dfnum;            (* identify s's dfnumber *)
         dflist[dfn] := s;      (* write s into the list using its
                                             dfnum for an index *)
         gph[prev].stnext := s;
         prev := s
      end
   end;
   gph[prev].stnext := 0;          (* zero indicates vertex prev is
                                           last to be st-numbered *)
   pltest(first);           (* test the component for planarity *)
   writeout(4)
end;  (* stnumber *)


function min (a,b : integer) : integer;
(* Function computes the minimum of two values *)
begin
   if a <= b then  min := a
   else  min := b
end;  (* min *)
```

```
procedure biconnect (v : integer);
(* This recursive procedure finds the biconnected components
   of a graph.  It also repositions (if needed) the two
   vertices in the record adjpoint so that NODE[1] holds the
   vertex you came from and NODE[2] holds the vertex you went
   to during the depth-first search.  The vertices' adjacency
   list pointers are switched accordingly.  Instead of a
   separate stack to hold new edges as they are encountered,
   the actual adjpoint records are "stacked" or linked.  This
   is done with the use of the pointer ELINK.  The procedure
   designates an edge to be either a "tree" or "back" edge.
   Depth-first search numbers and low point values are also
   assigned.  The local variable W represents an adjacent
   point.  TEMPTR and TEMP are used in the switching process.
   LOC tells the location of vertex v at any time.  BPTR
   points to adjacent nodes *)
var w,loc,temp : integer;   temptr,bptr : ptr;
begin
   count := count + 1;
   gph[v].flag := 'u';              (* mark the vertex as "used" *)
   gph[v].dfnum := count;         (* assign a depth-first number *)
   gph[v].lowpt := count;            (* assign a low point value *)
   bptr := gph[v].adjlist;     (* set bptr pointer to an adjacent
                                                      node of v *)

   while bptr ~= nil do
   (* while not at end of adjacency list for v *)
   begin
      if v = bptr^. node[1] then       (* if node[1] is vertex v *)
      begin
         w := bptr^.node[2];              (* then node[2] holds an
                                                 adjacent point *)
         loc := 1                        (* v is in location one *)
      end
      else
      begin
         w := bptr^.node[1];                    (* else vice versa *)
         loc := 2
      end;
      if gph[w].flag = 'a' then    (* is the point "available" *)
      begin
         if loc = 2 then                  (* if v is in node[2] *)
         begin                     (* then switch the vertices *)
            temp := bptr^.node[1];
            bptr^.node[1] := bptr^.node[2];
            bptr^.node[2] := temp;

            temptr := bptr^.next[1];     (* switch the pointers *)
            bptr^.next[1] := bptr^.next[2];
            bptr^.next[2] := temptr;

            loc := 1
         end;
         bptr^.elink := top;            (* add edge to edge stack *)
```

```
        top := bptr;              (* move pointer up one position *)
        bptr^.edgetype := 't';       (* mark edge as "tree" edge *)

        biconnect(w);                     (* start a search from w *)

        if gph[w]. lowpt < gph[v].dfnum then
        (* check for articulation points *)
           gph[v].lowpt := min(gph[w].lowpt,gph[v].lowpt)
        else
        (* an articulation point has been found *)
        begin
           bcomp := bcomp + 1;
           stnumber(v,w)
        end
     end
     else
        if bptr^.edgetype = 'n' then  (* is the edge "neither" *)
        begin
           if gph[bptr^.node[1]].dfnum >
                 gph[bptr^.node[2]].dfnum then
           (* check which node was visited first *)
           begin
              if v = bptr^.node[1] then
                 loc := 2
              else  loc := 1;
              temp := bptr^.node[1];     (* switch the vertices *)
              bptr^.node[1] := bptr^.node[2];
              bptr^.node[2] := temp;

              temptr := bptr^.next[1];  (* switch the pointers *)
              bptr^.next[1] := bptr^.next[2];
              bptr^.next[2] := temptr
           end;
           bptr^.elink := top;              (* add edge to stack *)
           top := bptr;         (* move pointer up in edge stack *)
           bptr^.edgetype := 'b';  (* mark edge as "back" edge *)

           gph[v].lowpt := min(gph[v].lowpt,gph[w].dfnum)
        end;
     bptr := bptr^.next[loc];        (* look at next point on v's
                                                  adjacency list *)
   end
end;  (* biconnect *)


procedure search;
(* Procedure finds an "available" start point for the search of
   a connected component.  If the adjacency list for an
   "available" node is empty, the number of single points is
   incremented by one.  Local variable T is used as a counter and
   and an index variable *)
var t : integer;
begin
   for t := 1 to total do
   begin
      if gph[t].flag = 'a' then    (* is the vertex "available" *)
         if gph[t].adjlist ~= nil then
         (* are there incident edges *)
         begin
            bcomp := 0;
            comp := comp + 1;
            top := nil;                   (* stack of edges is empty *)
            writeout(3);                          (* print message 3 *)
            count := 0;
            biconnect(t)               (* start search for biconnected
                                          component from vertex t *)
         end
         else   singlepts := singlepts + 1
   end
end;  (* search *)


(*********************************************************************
                        start of main program
*********************************************************************)

begin
   chr := ' ';
   while chr ~= '.' do                  (* while not at end of data *)
   begin
      readln(total);                (* read the number of vertices *)
      createvertices;                  (* create list of vertices *)
      readandbuild;                    (* read input graph and form
                                                  adjacency lists *)
      for d := 1 to max do
         dflist[d] := 0;
      singlepts := 0;
      comp := 0;
      count := 0;
      writeout(1);                            (* print message one *)
      search;          (* start search for a connected component *)
      writeout(2)                               (* print message 2 *)
   end
end  (* main program *)
```