EFFICIENT ALGORITHMS FOR GRAPH MANIPULATION

BY

JOHN HOPCROFT

ROBERT TARJAN

STAN-CS-71-207

MARCH, 1971

COMPUTER SCIENCE DEPARTMENT

School of Humanities and Sciences

STANFORD UNIVERSITY

Efficient Algorithms for Graph Manipulation

John Hopcroft

Robert Tarjan

Stanford University, Stanford, California

Abstract: Efficient algorithms are presented for partitioning a graph into connected components, biconnected components and simple paths. The algorithm for partitioning of a graph into simple paths is iterative and each iteration produces a new path between two vertices already on paths. (The start vertex can be specified dynamically.) If V is the number of vertices and E is the number of edges each algorithm requires time and space proportional-to max(V,E) when executed on a random access computer.

This research was supported by the Hertz Foundation and by the Office of Naval Research under grant number N-00014-67-A-0112-0057 NR 044-402. Reproduction in whole or in part is permitted for any purpose of the United States Government.

EFFICIENT ALGORITHMS FOR GRAPH MANIPULATION

John Hopcroft
Robert Tarjan
Stanford University, Stanford, California

Graphs arise in many different contexts where it is necessary to represent interrelations between data elements. Consequently algorithms are being developed to manipulate graphs and test them for various properties. Certain basic tasks are common to many of these algorithms. For example, in order to test a graph for planarity, one first decomposes the graph into biconnected components and tests each component separately. If one is using an algorithm [4] with asymptotic growth of $V \log V$ to test for planarity, it is imperative that one use an algorithm for partitioning the graph whose asymptotic growth is linear with the number of edges rather than quadratic in the number of vertices. In fact, representing a graph by a connection matrix in the above case would result in spending more time in constructing the matrix than in testing the graph for planarity if it were represented by a list of edges. It is with this in mind that we present a structure for representing graphs in a computer and several algorithms for simple operations on the graph. These include dividing a graph into connected components, dividing a graph into biconnected components, and partitioning a graph into simple paths. The algorithm for division into connected components is well-known [6]. The other two algorithms are original. For a graph with V vertices and E edges, each algorithm requires time and space proportional to $\max(V,E)$.

Standard graph terminology will be used throughout this discussion. See for instance [2]. We assume that the graph is initially given as a list of pairs of vertices, each pair representing an edge of the graph. The order of the vertices is unimportant; that is, the graph is unordered. Labels may be attached to some or all of the vertices and edges.

Our model is that of a random-access computer with standard operations; accessing a number in storage requires unit time. We allow storage of numbers no larger than $k \max(V,E)$ where k is some constant. (If the labels are large data items, we will assume that they are numbered with small integer codes and referred to by their codes; there are no more than $k \max(V,E)$ labels.) It is easy to see and may be proved rigorously that most interesting graph procedures require time at least proportional to E when implemented on any reasonable model of a computer, if the input is a list of edges. This follows from the fact that each edge must be examined once.

It is very important to have an appropriate computer representation for graphs. Many researchers have described algorithms which use the matrix representation of a graph [1]. The time and space bounds for such algorithms generally are at least V^2 [3] which is not as small as possible if E is small. (In planar graphs for instance, $E \leq 3V-6$.) We use a list structure representation of a graph. For each vertex, a list of vertices to which it is adjacent is made. Note that two entries occur for each edge, one for each of its end points. A cross-link between these two entries is often useful. Note also that a directed graph may be represented in this fashion; if vertex v_2 is on the list of vertices adjacent to v_1 , then (v_1,v_2) is a directed edge of the graph. Vertex v_1 is called the tail, and vertex v_2 is called the head of the edge.

A directed representation of an undirected graph is a representation of this form in which each edge appears only once; the edges are directed according to some criterion such as the direction in which they are transversed during a search. Some version of this structure representation is used in all the algorithms.

One technique has proved to be of great value. That is the notion of search, moving from vertex to adjacent vertex in the graph in such a way that all the edges are covered. In particular depth-first search is the basis of all the algorithms presented here. In this pattern of search, each time an edge to a new vertex is discovered, the search is continued from the new vertex and is not renewed at the old vertex until all edges from the new vertex are exhausted. The search process provides an orientation for each edge, in addition to generating information used in the particular algorithms.

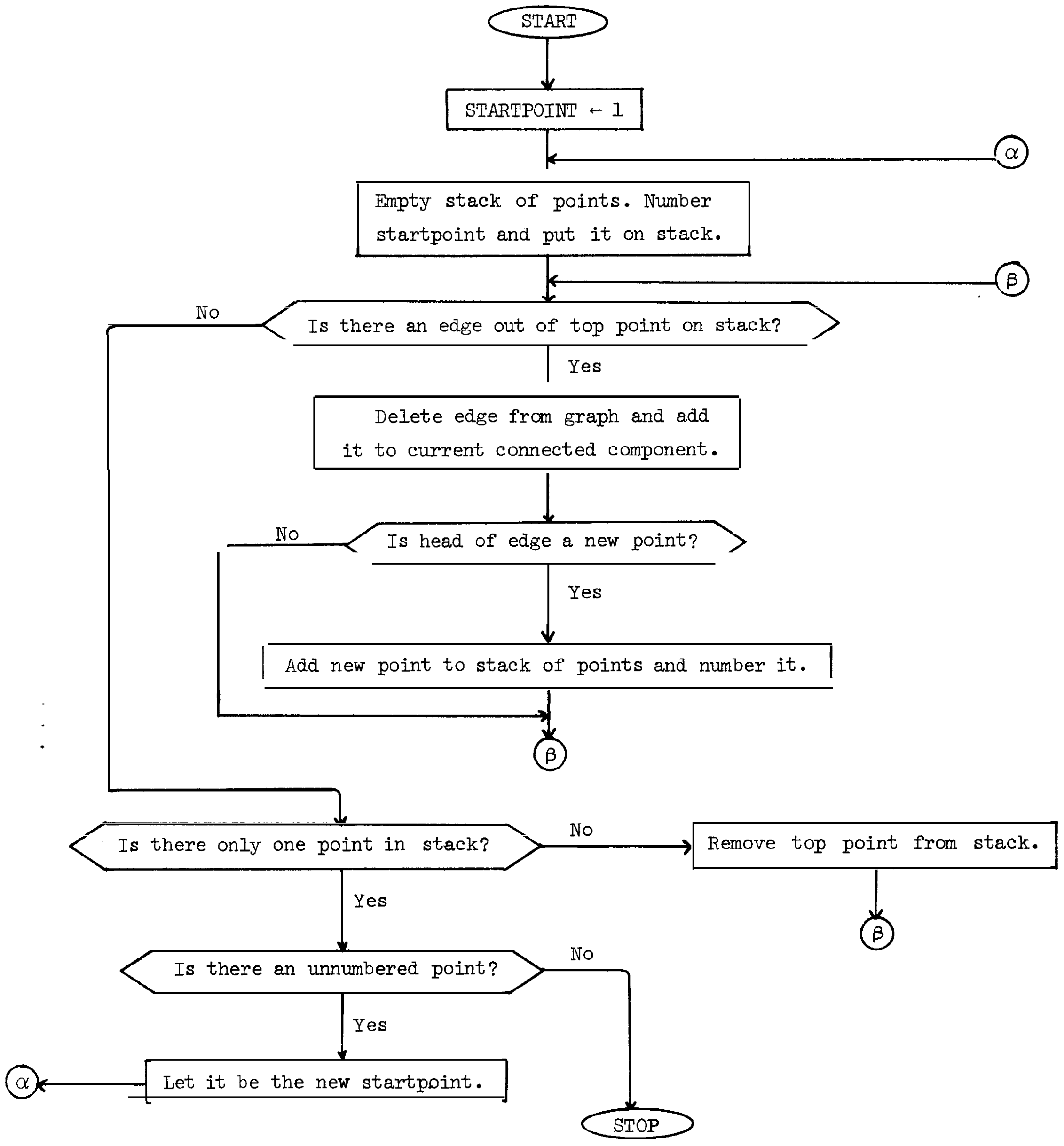
Detailed Description of the Algorithms

Algorithm for Finding the Connected Components of a Graph

This algorithm finds the connected components of a graph by performing depth-first search on each connected component. Each new vertex reached is marked. When no more vertices can be reached along edges from marked vertices, a connected component has been found. An unmarked vertex is then selected, and the process is repeated until the entire graph is explored.

The details of the algorithm appear in the flowchart (Figure 1). Since the algorithm is well-known, and since it forms a part of the algorithm for finding biconnected components, we omit proofs of its correctness and time bound. These proofs may be found as part of the proofs for the biconnected components algorithm. The algorithm requires space proportional to $\max(V,E)$ and time proportional to $\max(V,E)$ where V is the number of vertices and E is the number of edges of the graph.

Figure 1: Flowchart for Connected Components Algorithm


START
STARTPOINT ← 1
α
Empty stack of points. Number startpoint and put it on stack.
β
No
Is there an edge out of top point on stack?
Yes
Delete edge from graph and add it to current connected component.
No
Is head of edge a new point?
Yes
Add new point to stack of points and number it.
β
Is there only one point in stack?
No
Remove top point from stack.
β
Yes
Is there an unnumbered point?
No
Yes
Let it be the new startpoint.
α
STOP

Algorithm for Finding the Biconnected Components of a Graph

This algorithm breaks a graph into its biconnected components by performing a depth-first search along the edges of the graph. Each new point reached is placed on a stack, and for each point a record is kept of the lowest point on the stack to which it is connected by a path of unstacked points. When a new point cannot be reached from the top of the stack, the top point is deleted, and the search is continued from the next point on the stack. If the top point does not connect to a point lower than the second point on the stack, then this second point is an articulation point of the graph. All edges examined during the search are placed on another stack, so that when an articulation point is found the edges of the corresponding biconnected component may be retrieved and placed in an output array.

When the stack is exhausted, a complete search of a connected component has been performed. If the graph is connected, the process is complete. Otherwise, an unreached node is selected as a new starting point and the process repeated until all of the graph has been examined. Isolated points are not listed as biconnected components, since they have no adjacent edges. They are merely skipped. The details of the algorithm are given in the flowchart (Figure 2). Note that this flowchart gives a non-deterministic algorithm, since any new edge may be selected in block A. The actual program is deterministic; the choice of an edge depends on the particular representation of the graph.

We will prove that the non-deterministic algorithm terminates on all simple graphs without loops, and we also derive a bound on the execution time. We will then prove the correctness of the algorithm, by induction on the number of edges in the graph. Note that the algorithm requires storage space proportional to max(V,E) , where V is the number of vertices and E is the number of edges of the graph.

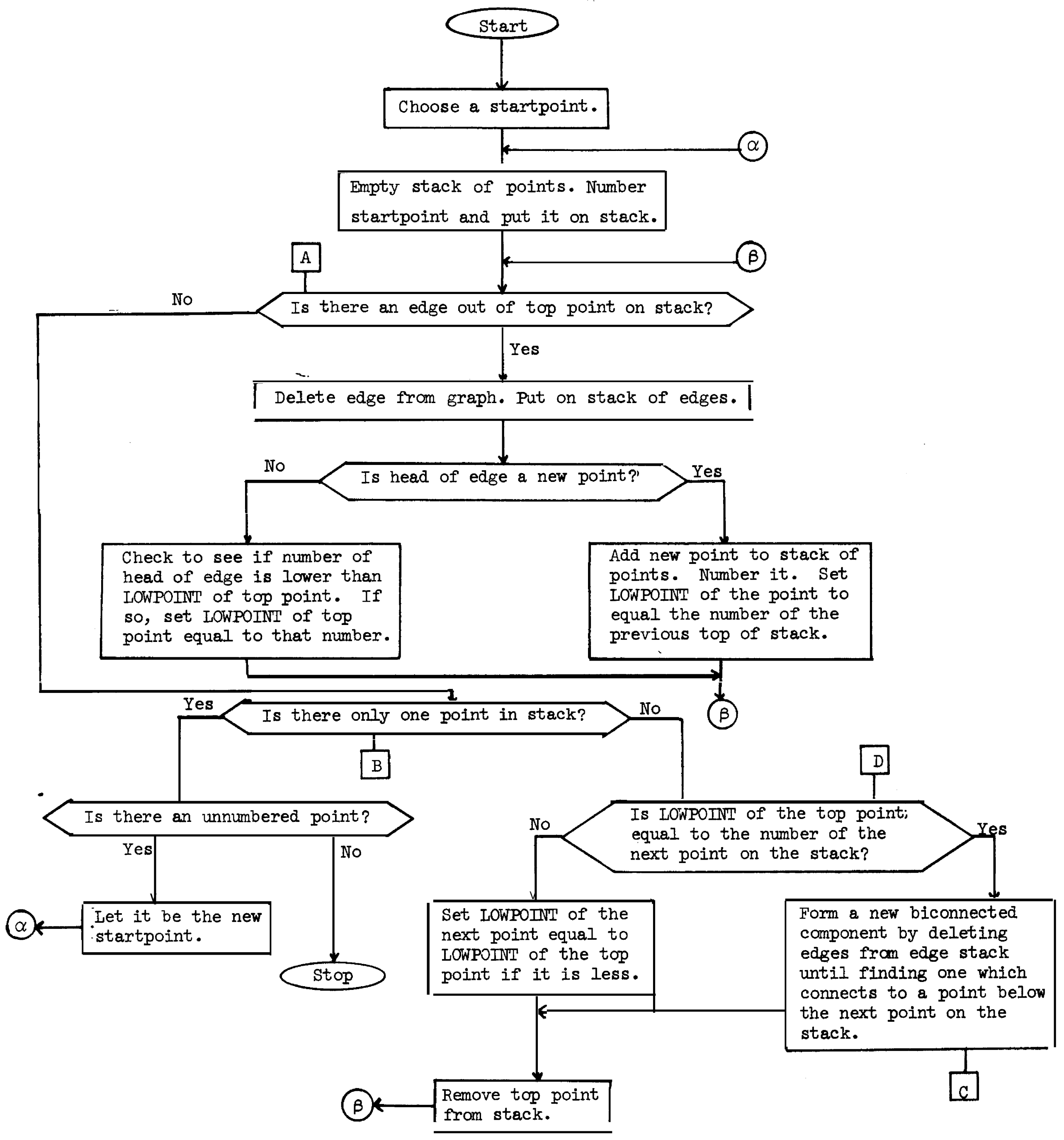
Let us consider applying the algorithm to a graph. Referring to the flowchart, every passage through the YES branch of block A causes an edge to be deleted from the graph. Each passage through the NO branch of block B causes a point to be deleted from the stack. Once a point is deleted from the stack it is never added to the stack again, since all adjacent edges have been examined. Each edge is deleted from the stack of edges once in block C. Thus the blocks directly below the YES branch of block A are executed at most E times, those below the NO branch of block B at most V times, and the total time spent in block C is proportional to E . Therefore there is some k such that for all graphs the algorithm takes no more than k max(V,E) steps. A more explicit time bound may be calculated by referring to the program.

Suppose the graph G contains no edges. By examining the flowchart we see that the algorithm, when applied to G, will terminate after examining each point once and listing no components. Thus the algorithm operates correctly in this case. Suppose the algorithm works correctly on all graphs with E-1 or fewer edges. Consider applying the algorithm to a graph G with E edges. Since the stack of points becomes empty at least once during the operation of the algorithm, and since the YES branch at block D must be taken when only two points are on the stack, every edge must not only be placed on the stack of edges but must be removed in block C. Consider the first time block C is reached when the algorithm is applied to graph G. Suppose not all the edges in the graph are removed from the stack of edges in this execution of block C. Then p , the second point on the stack, is an articulation point and separates the removed edges from the other edges in the graph.

Consider only the set of removed edges. If the algorithm is applied to the subgraph G' of G made up of these edges, with p used as the start point, then the steps taken are the same as those taken during the analysis of the edges of G' when the input is the entire graph. Since G' contains fewer edges than G , the algorithm operates correctly on G' . G' must be biconnected, since otherwise block C is reached before G' is completely examined, contrary to our assumption that block C is reached for the first time only after all edges of G' are examined. If we delete the set of edges of G' from G , we get another graph G" with fewer edges than G . The algorithm operates correctly on G" by assumption. The behavior of the algorithm on G is simply a composite of its behavior on G' and G" ; thus the algorithm must operate correctly on G.

Now suppose that the first time block C is reached, all the edges of G are removed from the stack of edges. We want to show that in this case G is biconnected. Suppose that G is not biconnected. Then choose a biconnected component of G which may be separated by removing some one point. Let the edges making up this component be subgraph G' of G ; let the remainder of G be G" . The algorithm operates correctly on G' and on G" by assumption. The behavior of the algorithm on G is a composite of its behavior on G' and on G" . But the algorithm reaches block C once while processing G' and at least once while processing G" . This contradicts the fact that the algorithm only reaches block C once while processing G . Thus G must be biconnected, and the algorithm operates correctly on G. By induction, the algorithm is correct for all simple graphs without loops.

Figure 2: Flowchart for Biconnected Components Algorithm


Start
Choose a startpoint.
α
Empty stack of points. Number startpoint and put it on stack.
A
β
Is there an edge out of top point on stack?
No
Yes
Delete edge from graph. Put on stack of edges.
Is head of edge a new point?
No
Yes
Check to see if number of head of edge is lower than LOWPOINT of top point. If so, set LOWPOINT of top point equal to that number.
Add new point to stack of points. Number it. Set LOWPOINT of the point to equal the number of the previous top of stack.
β
Is there only one point in stack?
Yes
No
B
Is there an unnumbered point?
Yes
No
Let it be the new startpoint.
α
Stop
D
Is LOWPOINT of the top point equal to the number of the next point on the stack?
No
Yes
Set LOWPOINT of the next point equal to LOWPOINT of the top point if it is less.
Form a new biconnected component by deleting edges from edge stack until finding one which connects to a point below the next point on the stack.
C
Remove top point from stack.
β

Algorithm for Finding Simple Paths in a Graph

This algorithm may be used to partition a graph into simple paths, such that all the paths exhaust the edges of the graph. Each iteration of the algorithm produces a new path which contains no vertex twice, and which connects the chosen startpoint with some other vertex which already occurs in a path. Total running time is proportional to the number of edges in the graph. The starting point for each successive path may be selected arbitrarily. In fact, the initial edge of each successive path may be selected arbitrarily from the set of unused edges.

The algorithm is highly dependent on the graph being biconnected. (The biconnected components of a graph are found using the previously described algorithm.) In order to find a new path, the initial edge is selected and the head of the edge is checked. If this point has never been reached before, a depth-first search is begun which must end in a path since the graph is biconnected. The search generates a tree-like structure; specifically, it is a tree with extra edges connecting some nodes with their (not necessarily immediate) ancestors. (We will visualize the tree drawn so that the root, which is an ancestor of all points, is at the bottom of the tree.) Enough information is saved from this tree so that if a point in it is reached when building another path, the path may be completed without any further search.

The flowchart (Figs. 3 and 4) gives the details of the algorithm. It is divided into two parts; one for the depth-first search process and one for path construction using previously gathered information. We shall prove the correctness of the algorithm and give a time bound for its operation. To derive the time bound, we assume that one point is marked old initially, and a different point is selected as the initial startpoint. The algorithm is then run repeatedly with arbitrary startpoints until all edges are used to form paths.

Let us consider path generation using depth-first search; that is, suppose the algorithm is applied and that the head of the first edge selected is previously unreached. Referring to the flowchart, we see that the search process is very similar to that used in the biconnectivity algorithm. A search tree is generated, and each edge examined is either part of the tree or connects a point to one of its predecessors in the tree. LOWPOINT is exactly the same as in the biconnectivity algorithm; it gives the number of the lowest point in the tree reachable from a given point by continuing out along the tree and taking one edge back toward the root. The forward edges point along this path, while the backward edges point back along the tree branches. We have shown in the correctness proof of the biconnectivity algorithm that, if the graph is biconnected, LOWPOINT of a given point must point to a node which is an ancestor of the immediate predecessor of the given point. In particular, LOWPOINT of the second point in the search tree must indicate an old point which is not the startpoint. Therefore the algorithm will find a path containing the initial edge. Note that all points encountered during the search process must either be old or unreached, since every point reached in a previous search either has had all its edges examined or has been included in a path.

Let us now suppose that the head of the first edge has been reached previously but is not marked old. Then the forward and backward pointers, along with the LOWPOINT values, allow the algorithm to construct a path without further search. First, if the number of the head of the edge is less than the number of the startpoint, then following backward pointers will certainly produce a simple path, since the root of a search tree must be old and each successive point along a backward path has a lower number and thus is distinct from the other points in the path. If the initial edge is part of a search tree and the startpoint is the predecessor of the second point, then LOWPOINT of the second point must be less than the number of the startpoint. Following forward edges until reaching a point numbered lower than the startpoint and then following backward edges, will produce a simple path. This is true since the forward edges point through descendants of the tree, with the single exception of the edge whose head is a point below startpoint in the tree. The last case to consider occurs when the initial edge is not part of a search tree but points from a node to one of its descendants in a tree. In this case some node in the tree between the startpoint and the second point of the path must have a LOWPOINT value less than the number of the startpoint. If we follow backward edges until the first such point is reached, then follow forward edges until a point numbered less than the startpoint is reached, and finally follow backward edges until an old point is reached, we will generate a simple path. Note that the first forward edge taken cannot lead to the previous point, because if it did the LOWPOINT value at the previous point would be less than the number of startpoint, and the forward edge from this point would have been chosen instead of the backward edge.

We thus see that each execution of the pathfinding algorithm produces a simple path, assuming that the algorithm is applied to a biconnected graph with at least one point which is not the first startpoint marked old initially. Since each edge is examined at most once in the search section of the algorithm, and since each edge is put into a path once, there is a constant k such that the time required to execute the algorithm until no edges are unused is less than kE steps, where E is the number of edges in the graph. (Note that the number of vertices, V , is less than E if the graph is biconnected.) Detailed examination of the program will produce a more exact time bound.

Another algorithm for finding simple paths exists. Lempel, Even, and Cederbaum[5] have described an algorithm for numbering the vertices of a biconnected graph such that: (i) each number is an integer in the range 1 to V , where V is the number of vertices on the graph; (ii) vertices 1 and V are joined by an edge; (iii) for all $1 < i < V$, vertex i is joined to at least two vertices, one with a higher number and one with a lower number. We may use this algorithm to partition a graph into simple paths.

Given a start point and an adjacent end point, number the vertices so that the start point is 1 , the endpoint is V , and the numbering satisfies the conditions above. Take edge (1,V) as the first path. Given an arbitrary start point, find an edge to a higher numbered vertex. Continue to find edges to successively higher numbered vertices until an old vertex is reached. If no edge to a higher numbered vertex exists from the start vertex, select edges to successively lower numbered vertices until an old vertex is reached.

This algorithm is clearly correct and looks conceptionally simpler. However, Lempel, Even, and Cederbaum present no efficient implementation of their numbering algorithm, and the only efficient way we have found to implement it requires using the previously described pathfinding algorithm in a more complicated form. Thus the new algorithm requires time and space proportional to max(V,E) , but the constants of proportionality are larger than those for the implemented algorithm.

Figure 3: Flowchart for Pathfinding Algorithm (I)

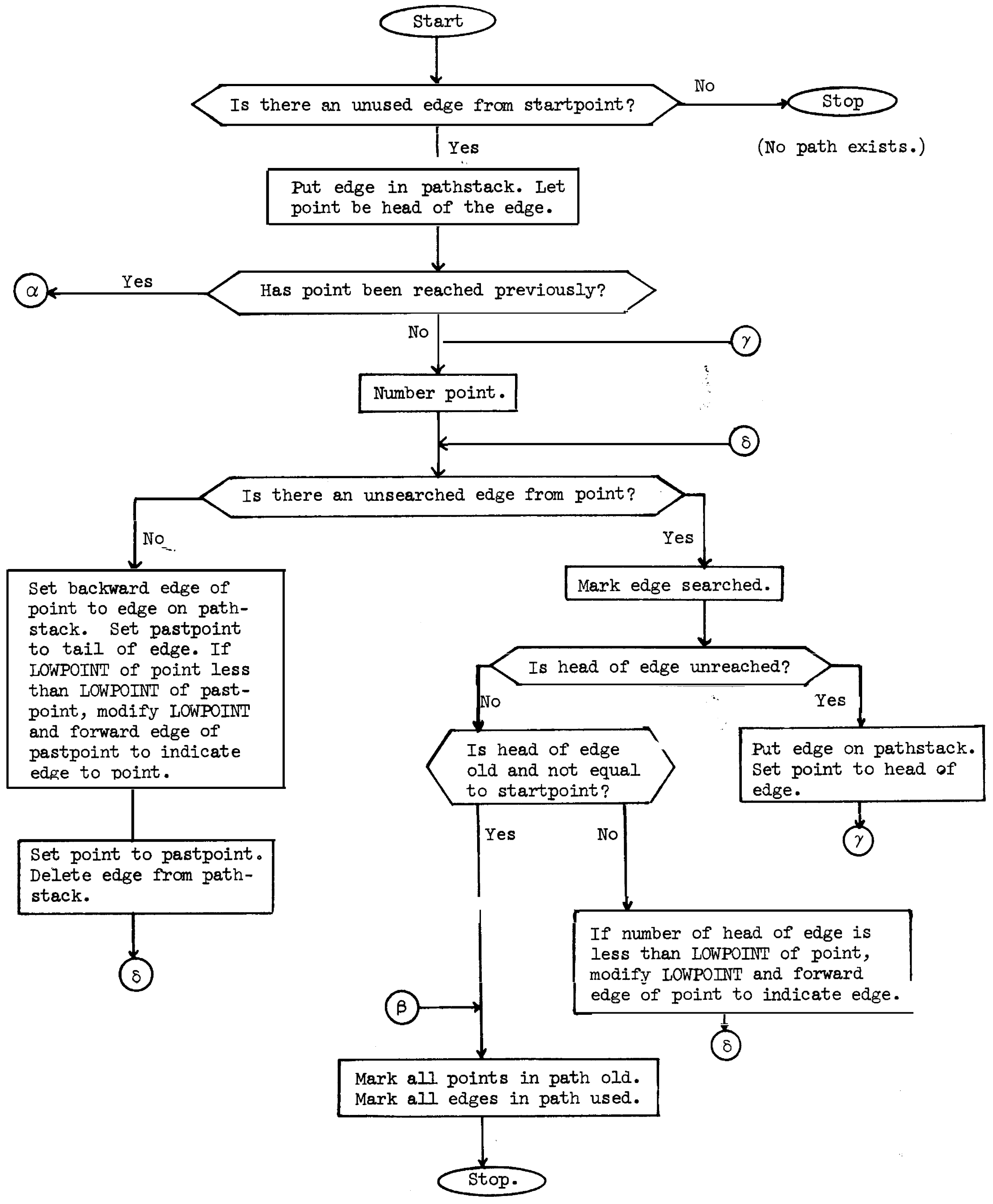
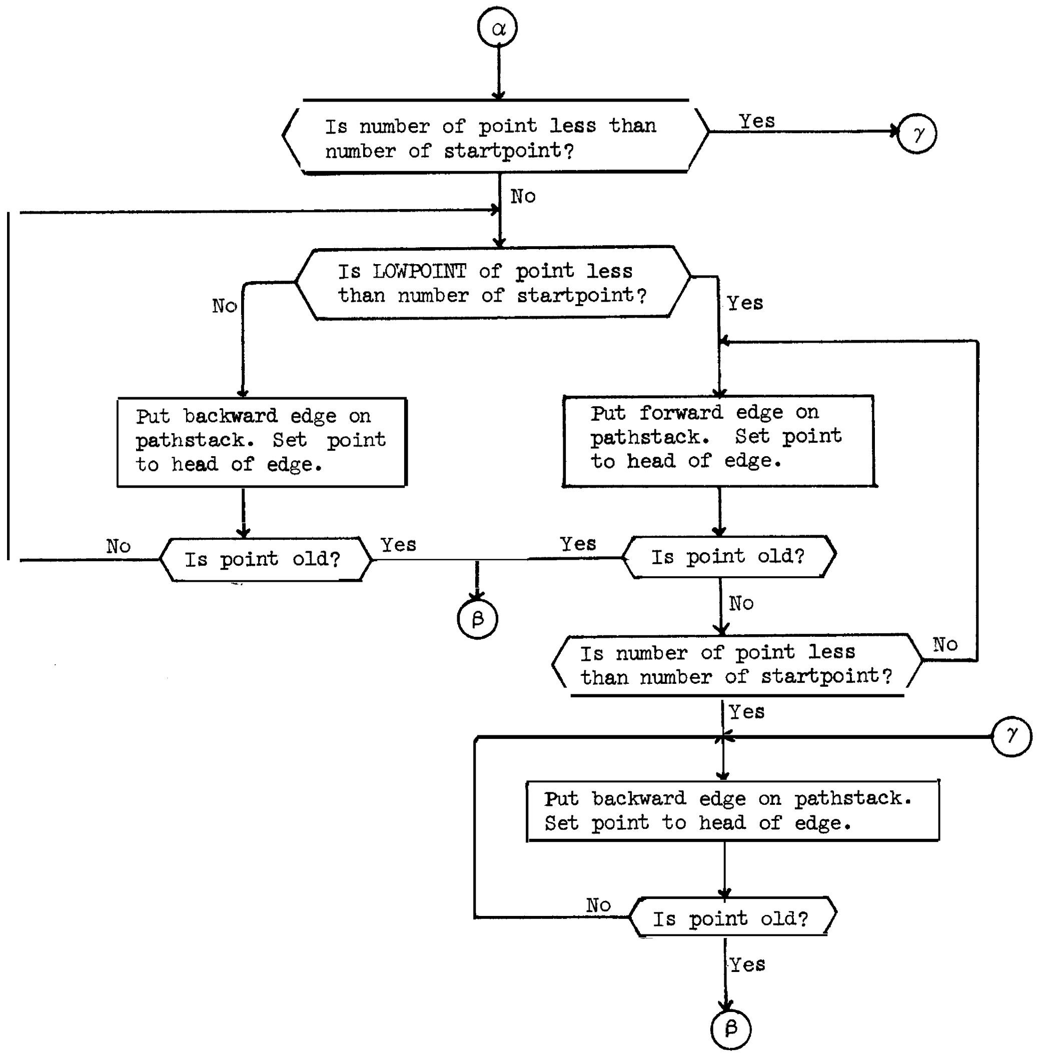

Start
Is there an unused edge from startpoint?
No
Stop
(No path exists.)
Yes
Put edge in pathstack. Let point be head of the edge.
Has point been reached previously?
Yes
α
No
γ
Number point.
δ
Is there an unsearched edge from point?
No
Yes
Set backward edge of point to edge on pathstack. Set pastpoint to tail of edge. If LOWPOINT of point less than LOWPOINT of pastpoint, modify LOWPOINT and forward edge of pastpoint to indicate edge to point.
Set point to pastpoint. Delete edge from pathstack.
δ
Mark edge searched.
Is head of edge unreached?
No
Yes
Is head of edge old and not equal to startpoint?
Put edge on pathstack. Set point to head of edge.
γ
Yes
No
If number of head of edge is less than LOWPOINT of point, modify LOWPOINT and forward edge of point to indicate edge.
β
δ
Mark all points in path old. Mark all edges in path used.
Stop.

Figure 4 Flowchart for Pathfinding Algorithm (II)


α
Is number of point less than number of startpoint?
Yes
γ
No
Is LOWPOINT of point less than number of startpoint?
No
Yes
Put backward edge on pathstack. Set point to head of edge.
Put forward edge on pathstack. Set point to head of edge.
No
Is point old?
Yes
Yes
Is point old?
β
No
Is number of point less than number of startpoint?
No
Yes
γ
Put backward edge on pathstack. Set point to head of edge.
No
Is point old?
Yes
β

Implementation

The algorithms for finding connected components, biconnected components, and simple paths were implemented in Algol using the Algol W compiler at Stanford University. Auxiliary subroutines were also implemented. Brief descriptions of the procedures are provided below.

ADD2(A,B,STACK,PTR) : This procedure adds value A followed by value B to the top of stack STACK and increments the pointer to the top of the stack (PTR). Stacks are represented as arrays; the top of the stack is the highest filled location.

NEXTLINK(POINT,VALUE): This procedure is used to build the structural representation of a graph. It adds VALUE to the list of vertices adjacent to POINT. (POINT,VALUE) is an edge (possibly directed) of the graph.

CONNECT(V,E,EPTR,EDGELIST,COMPONENTS): This procedure, given a graph with V vertices and E edges, whose edges are listed in EDGELIST, computes the connected components of the graph and places the edges of the components in COMPONENTS. Each component is preceded by an entry containing the number of edges E' of the component. The edges are oriented for output according to the direction in which they were searched (head first, tail second).

BICONNECT(V,E,EPTR,EDGELIST,BICOMPONENTS): This procedure, given a graph with V vertices and E edges, whose edges are listed in EDGELIST, computes the biconnected components of the graph and places them in BICOMPONENTS. Each component is preceded by an entry containing the number of edges E" of the component. The edges are oriented for output according to the direction in which they were searched (head first, tail second).

PATHFINDER(STARTPT,PATHPT,CODEVALUE,PATH): This procedure, given a list structure representation of a biconnected graph with certain vertices marked as old, constructs a simple path from STARTPOINT to some old vertex, saving information to be used in constructing succeeding paths. The new path is stored in array PATH. Calling PATHFINDER repeatedly may be used to partition the graph into simple paths.

Further comments may be found in the program listings, which follow.

```
PROCEDURE ADD2(INTEGER VALUE A,B;INTEGER ARRAY STACK(*);
    INTEGER VALUE RESULT PTR);
  BEGIN
    COMMENT ***********************************************************
    *      PROCEDURE TO ADD VALUES A, B TO STACK "STACK" AND
    *      INCREASE STACK POINTER "PTR" BY 2.
    **************************************************************;
    PTR:=PTR+2;
    STACK(PTR-1):=A;
    STACK(PTR):=B
  END;
```

```
PROCEDURE NEXTLINK( INTEGER VALUE POINT,VAL);
  BEGIN
    COMMENT ***********************************************************
    *      PROCEDURE TO ADD DIRECTED EDGE (POINT,VAL) TO
    *      STRUCTURAL REPRESENTATION OF A GRAPH.
    *
    *      GLOBAL VARIABLES:
    *        HEAD(V+1::V+2*E),NEXT(1::V+2*E): STRUCTURAL
    *          REPRESENTATION OF THE GRAPH.
    *        FREENEXT: CURRENT LAST ENTRY IN NEXT ARRAY.
    **************************************************************;
    FREENEXT:=FREENEXT+1;
    NEXT(FREENEXT):=NEXT(POINT);
    NEXT(POINT):=FREENEXT;
    HEAD(FREENEXT):=VAL
  END;
```

```
INTEGER PROCEDURE MIN(INTEGER VALUE A,B);
  COMMENT *************************************************************
  *      PROCEDURE TO COMPUTE THE MINIMUM OF TWO INTEGERS.
  ****************************************************************;
  IF A<B THEN A ELSE B;
```

```
PROCEDURE CONNECT(INTEGER VALUE V,E; INTEGER RESULT CPTR;
    INTEGER ARRAY EDGELIST,COMPONENTS(*));
   BEGIN
    COMMENT ***********************************************************
    *      PROCEDURE TO FIND THE CONNECTED COMPONENTS OF A
    *      GRAPH.
    *
    *      PARAMETERS:
    *        V,E: INPUT NUMBER OF VERTICES AND EDGES OF THE
    *          GRAPH.
    *        EDGELIST(1::2*E): INPUT LIST OF EDGES OF GRAPH.
    *        COMPONENTS(1::3*E): OUTPUT LIST OF EDGES OF
    *          COMPONENTS FOUND.  EACH COMPONENT IS PRECEDED BY
    *          AN ENTRY GIVING THE NUMBER OF EDGES OF THE
    *          COMPONENT.
    *        CPTR: OUTPUT POINTER TO LAST ENTRY IN COMPONENTS.
    *
    *      GLOBAL VARIABLES:
    *        HEAD(V+1::V+2*E),NEXT(1::V+2*E): STRUCTURAL
    *          REPRESENTATION OF THE GRAPH (UNDIRECTED, NO
    *          CROSS-LINKS).
    *        FREENEXT: LAST ENTRY IN NEXT ARRAY.
    *
    *      LOCAL VARIABLES:
    *        NUMBER(1::V+1): ARRAY FOR NUMBERING THE VERTICES
    *          DURING DEPTH-FIRST SEARCH.
    *        CODE: CURRENT HIGHEST VERTEX NUMBER.
    *        POINT: CURRENT POINT BEING EXAMINED DURING SEARCH.
    *        V2: NEXT POINT TO BE EXAMINED DURING SEARCH.
    *        OLDPTR: POSITION IN COMPONENTS TO PLACE E VALUE OF
    *          NEXT COMPONENT.
    *
    *      GLOBAL PROCEDURES:
    *        ADD2,NEXTLINK.
    *
    *      A RECURSIVE DEPTH-FIRST SEARCH PROCEDURE IS USED TO
    *      EXAMINE CONNECTED COMPONENTS OF THE GRAPH.
    ***********************************************************;
    INTEGER ARRAY NUMBER(1::V+1);
    INTEGER CODE,POINT,V2,OLDPTR;
    PROCEDURE CONNECTOR(INTEGER VALUE POINT, OLDPT);
      COMMENT ********************************************************
      *      RECURSIVE PROCEDURE TO FIND A CONNECTED COMPONENT,
      *      USING DEPTH-FIRST SEARCH.
      *
      *      PARAMETERS:
      *        POINT: STARTPOINT OF SEARCH.
      *        OLDPT: PREVIOUS STARTPOINT.
      *
      *      GLOBAL VARIABLES:
      *        SEE CONNECT FOR DESCRIPTION.
      *
      *      GLOBAL PROCEDURES:
      *        ADD2.
      *
```

```
*     EXAMINE EACH EDGE OUT OF POINT.
*********************************************************;
WHILE NEXT(POINT)>0 DO
  BEGIN
    COMMENT **********************************************
    *     V2 IS HEAD OF EDGE.  DELETE EDGE FROM
    *     STRUCTURAL REPRESENTATION.
    *****************************************************;
    V2:=HEAD(NEXT(POINT));
    NEXT(POINT):=NEXT(NEXT(POINT));
    COMMENT **********************************************
    *     HAS THE EDGE BEEN SEARCHED IN THE OTHER
    *     DIRECTION?  IF SO, LOOK FOR ANOTHER EDGE.
    *****************************************************;
    IF (NUMBER(V2)<NUMBER(POINT))AND(V2¬=OLDPT) THEN
      BEGIN
        COMMENT ******************************************
        *     ADD EDGE TO COMPONENTS.
        *************************************************;
        ADD2(POINT,V2,COMPONENTS,CPTR);
        COMMENT ******************************************
        *     HAS A NEW POINT BEEN FOUND?
        *************************************************;
        IF NUMBER(V2)=0 THEN
          BEGIN
            COMMENT **************************************
            *     NEW POINT FOUND.  NUMBER IT.
            *********************************************;
            NUMBER(V2):=CODE:=CODE+1;
            COMMENT **************************************
            *     INITIATE A DEPTH-FIRST SEARCH FROM THE
            *     NEW POINT.
            *********************************************;
            CONNECTOR(V2,POINT);
          END
      END;
  END;
COMMENT **********************************************
*     CONSTRUCT THE STRUCTURAL REPRESENTATION OF THE
*     GRAPH.
*****************************************************;
FREENEXT:=V;
FOR I:=1 UNTIL V DO NEXT(I):=0;
FOR I:=1 UNTIL E DO
  BEGIN
    COMMENT ******************************************
    *     EACH EDGE OCCURS TWICE, ONCE FOR EACH
    *     ENDPOINT.
    *************************************************;
    NEXTLINK(EDGELIST(2*I-1),EDGELIST(2*I));
    NEXTLINK(EDGELIST(2*I),EDGELIST(2*I-1));
  END;
COMMENT **********************************************
*     INITIALIZE VARIABLES FOR SEARCH.
*****************************************************;
CPTR:=0;
POINT:=1;
FOR I:=1 UNTIL V+1 DO NUMBER(I):=0;
WHILE POINT<=V DO
  BEGIN
```

```
        COMMENT *********************************************
        *       EACH EXECUTION OF CONNECTOR SEARCHES A
        *       CCNNECTED CCMPCNENT.  AFTER EACH SEARCH,
        *       FINC A h UNNUMBEREC VERTEX AND SEARCH AGAIN.
        *       REPEAT UNTIL ALL VERTICES ARE INVESTIGATED.
        **********************************************;
        NUMBER(PCINT):=CCDE:=1;
        OLDPTR:=CPTR:=CPTR+1;
        CCNNECTCR(POINT,0);
        CCMMENT *********************************************
        *       COMPUTE NUMBER OF EDGES Of COMPONENT.
        **********************************************;
        COMPONENTS(OLDPTR):=(CPTR-OLDPTR)DIV 2 ;
        WHILE NUMBER(PCINT)¬=0 D O POINT:=POINT+1;
      END
  END;



PROCEDURE BICONNECT(INTEGER VALUE V,E; INTEGER RESULT RPTR;
    INTEGER ARRAY EDGELIST,BICOMPONENTS(*)) ;
  BEGIN
    COMMENT ***************************************************
    *       PROCEDURE TO FIND THE BICONNECTED COMPONENTS OF A
    *       GRAPH.
    *
    *       PARAMETERS:
    *          V,E: INPUT NUMBER OF VERTICES AND EDGES OF THE
    *             GRAPH.
    *          EDGELIST(1::2*E): INPUT LIST OF EDGES OF GRAPH.
    *          BICOMPCNENTS(1::3*E): OUTPUT LIST OF EDGES OF
    *             COMPONENTS FOUND.  EACH COMPONENT IS PRECEDED BY
    *             A N ENTRY GIVING THE NUMBER OF EDGES OF THE
    *             COMPONENT.
    *          BPTR:OUTPUT POINTER TO LAST ENTRY OF BICCMPONENTS.
    *
    *       GLOBAL VARIABLES:
    *          HEAD(V+1::V+2*E),NEXT(1::V+2*E): STRUCTURAL REPRE-
    *             SENTATION OF THE GRAPH (UNDIRECTED, N O CROSS-
    *             LINKS),
    *          FREENEXT: CAST ENTRY IN NEXT ARRAY.
    *
    *       LCCAL VARIABLES:
    *          NUMBER(1::V+1): ARRAY FCR NUMBERING THE VERTICES
    *             DURING DEPTH-FIRST SEARCH.
    *          CODE:CURRENT HIGHEST VERTEX NUMBER.
    *          EDGESTACK(1::2*E): STORAGE FOR LIST OF EDGES
    *             EXAMINE0 CURING SEARCH.
    *          EPTR: POINTER TO LAST ENTRY IN EDGESTACK.
    *          POINT: CURRENT POINT 8EING EXAMINED WRING SEARCH.
    *          V2: NEXT PC? NT TO BE EXAMINED DURING SEARCH.
    *          NEWLOWPT:LOWPOINT FCR BICONNECTED PART OF GRAPH
    *             ABOVE ANC INCLUDING V2.
*       OLDPTR: POSITION IN BICOMPONENTS TO PLACE E VALUE
*          OF NEXT COMPONENT.
*
*       GLOBAL PROCEDURES:
*         MIN,ACC2,NEXTLINK.
*
*       A RECURSIVE CEPTH-FIRST SEARCH PROCEDURE IS USED TO
*       DIVIDE THE GRAPH. THE LOWEST POINT REACHABLE FROM THE
*       CURRENT POINT WITHOUT GOING THROUGH PREVIOUSLY
*       SEAQCHEC PCINTS IS CALCULATED. THIS INFORMATION
*       ALLOWS DETERMINATION CF THE ARTICULATION POINTS AND
*       CIVISION C  F THE GRAPH.
********************************************************;
INTEGER ARRAY NUMBER(1::V+1);
INTEGER ARRAY EDGESTACK(1::2*E);
INTEGER CODE, EPTR,POINT ,V2,NEWLOWPT,OLDPTR ;
PROCEDURE BICCNNECTOR(INTEGER VALUE QESUCT POINT,CLDPT,
     LOWPO INT);
  CCMMENT ***********************************************
  *       RECURSIVE PROCEDURE TO SEARCH A CONNECTED COMPONENT
  *       AND FIND ITS BICONNECTED CQMPONENTS USING DEPTH-
  *       FIRST SEARCH.
  *
  *       PARAMETERS:
  *         POINT: STARTPOINT CF SEARCH, UNCHANGED CURING
  *            EXECUT ION.
  *         CLDPT: PFEVICUS STARTPOINT, UNCHANGED DURING
  *            EXECUTION.
  *         LOWPOINT: OUTPUT OF LOWEST POINT REACHABLE ON A
  *            PATH FCUND DURING SEARCH FORWARD.
  *
  *       GLOBAL VARIABLES:
  *         SEF BICONNECT FOR DESCRIPTION.
  *
  *       GLOBAL PROCECURES:
  *         MIN,ADD2.
  *
  *       EXAMINE EACH EDGE GUT CF PCINT.
  ******************************************************;
  WHILE NEXT(PCINT)>0 C O
    BEGIN
    COMMENT ********************************************
    *       V2 IS HEACOF THE EDGE. DELETE EDGE FROM
    *       STRUCTURAL REPRESENTATION.
    ****************************************************;
    V2:=HEAC(NEXT(PCINT));
    NEXT(POINT):=NEXT(NEXT(POINT));
    COMMENT ********************************************
    *       HAS. THE EDGE BEEN SEARCHED IN THE CTHER
    *       CIRECT ION?  IF SO, LOOK FOR ANOTHER EDGE.
    ****************************************************;
    IF(NUMBER(V2)<NUMBER(POINT))AND(V2¬=OLDPT) THEN
      BEGIh
      COMMENT ******************************************
      *       ADCECGE TO EDGESTACK.
      **************************************************;
      ADD2(POINT,V2,EDGESTACK,EPTR);
      COMMENT ******************************************
      *       HAS A NEW POINT BEEN FOUND?
      **************************************************;
```

```
IF NUMBER(V2)=0 THEN
  BEGIN
    CCMMENT ****************************************
    *      NEW POINT FOUND.   NUMBER IT.
    *********************************************;
    NUMBER(V2):=CODE:=CODE+1;
    CCMMENT ****************************************
    *      INITIATE A DEPTH-FIRST SEARCH FROM THE
    *      NEW POINT.
    *********************************************;
    NEWLCWPT:=V+1;
    BICONNECTOR(V2,POINT,NEWLOWPT);
    CCMMENT ****************************************
    *      NOTE THAT ALTHOUGH GLOBAL VARIABLE V2
    *      IS CHANGEC, ITS VALUE IS RESTCRFD UPON
    *      EXIT FROM THIS PROCEDURE.   RECALCULATE
    *      LOWPOINT.
    *********************************************;
    LOWPOINT:=MIN(LOWPOINT,NEWLOWPT);
    CCMMENT ****************************************
    *      IS POINT AN ARTICULATION POINT OF THE
    *      GRAPH?
    *********************************************;
    IF NEWLOWPT>=NUMBER(POINT) THEN
      BEGIN
        CCMMENT ***********************************
        *      POINT IS AN ARTICUCATICN POINT.
        *      OUTPUT EDGES OF COMPONENT FROM
        *      EDGESTACK.
        *****************************************;
        OLDPTR:=BPTR:=BPTR+1;
        WHILE NUMBER(EDGESTACK(EPTR-1))>NUMBER
            (PCINT) DO
          BEGIN
            ADC2(EDGESTACK(EPTR-1),EDGESTACK
                (EPTR),BICOMPONENTS,BPTR);
            EPTR:=EPTR-2
          END;
        COMMENT ***********************************
        *      ADD LAST EDGE.
        *****************************************;
        ADD2(POINT,V2,BICOMPONENTS,BPTR);
        EPTR:=EPTR-2;
        COMMENT ***********************************
        *      COMPUTE NUMBER OF EDGES OF
        *      COMPCNENT.
        *****************************************;
        BICCMPONENTS(OLDPTR):=(BPTR-OLDPTR)DIV 2 ;
      END
  END
ELSE
  COMMENT *******************************************
  *    NEW POINT NOT FOUND.   RECALCULATE LOWPOINT.
  ************************************************;
  LOWPOINT:=MIN(LOWPCINT,NUMBER(V2))
END
END;
COMMENT ***************************************************
*      CONSTRUCT THE STRUCTURAL REPRESENTATION Of THE GRAPH.
*********************************************************;
```

```
FREENEXT:=V;
FGR I:=1 UNTIL V DC NEXT(I):=0;
FOR I:=1 UNTIL E D C
  BEGIN
    COMMENT ************************************************
    *       EACH EDGE OCCURS TWICE, ONCE FOR EACH ENDPOINT.
    ************************************************;
    NEXTLINK(EDGELIST(2*I-1),EDGELIST(2*I));
    NEXTLINK(EDGELIST(2*I),EDGELIST(2*I-1))
  END;
CCMMENT ****************************************************
*       INITIALIZE VARIABLES FCR SEARCH,
***********************************************************;
EPTR:=0;
BPTR:=0;
PCINT:=1;
V2:=0;
FCP I:=1 UNTIL V+1 DO NUMBER(I):=0;
WHILE POINT<=V CC
  BEGIN
    CCMMENT ************************************************
    4       EACH EXECUTION Of BICONNECTOR SEARCHES A
    *       CONNECTEC COMPONENT OF THE GRAPH.  AfTER EACH
    *       SEARCH, FIND 4N UNNUMBERED VERTEX ANG SEARCH
    *       AGAIN.  REPEAT UNTIL ALL VERICES ARE EXAMINED.
    ************************************************;
    NUMBER(POINT):=CODE:=1;
    NEWLOWPT:=V+1;
    BICONNECTOR(PCINT,V2,NEWLOWPT);
    WHILE NUMBER(POINT)¬=0 DO POINT:=POINT+1
  END;
END;


  PROCEDURE PATHFINDER(INTEGER VALUE RESULT STARTPCINT,
      PATHPT,CCDEVALUE; INTEGER ARRAY PATH(*));
    BEGIN
      COMMENT **********************************************
      *       PROCECURE TO FIND DISJOINT PATHS WITH
      *       ARBITRARY STARTING POINTS IN A BICONNECTED
      *       GRAPH.  THE PCINTS OF EACH PATH ARE LISTED
      *       IN ARRAY "PATH".  THE FOLLOWING VARIABLES ARE
      *       ASSUMED GLOBAL:
      *       NEXT(1::V+2*E),HEAD(V+1,V+2*E),
      *       LINK(V+1,V+2*E) CEFINE THE GRAPH USING SINGLY
      *       LINKED EDGE LISTS AND A SET OF CRCSS REFERENCE
      *       PCINTERS.
      *       CLD(1::V),MARK(V+1,V+2*E) INDICATE USED
      *       PCINTS AND EDGES.
      *       PATHCCDE(1::V)IS THE CONSECUTIVE NUMBERING
      *       OF THE POINTS.
      *       LOWPCINT(1::V),FCRWARD(1::V),BACK(1::V) GIVE
      *       INFORMATICN SAVED FROM DEPTH-FIRST SEARCH.
      4       NODE(1::V) GIVES THE NEXT UNSEARCHED EDGE
      *       FRCM EACH POINT.
      **********************************************;
```

```
                   INTEGER PCINT, PASTEOGE, EDGE, PASTPOINT, V2;
                   PATH(1):= STAR TPOINT;
                   COMMENT *********************************************
                   *      CHOOSE INITIAL EDGE.
                   *********************************************;
                   EDGE:= NEXT(STARTPOINT);
                   WHILE(EDGE¬=0)AND MARK(EDGE) DO EDGE:=NEXT(EDGE);
                   IF EDGE=0 THEN
                     BEGIN
                       COMMENT *************************************
                       *      NO UNUSED ECGE ANDTHUS NO PATH EXISTS.
                       *************************************;
                       NEXT(STARTPOINT):=0;
                       PATHPT:=0 ;
                       GO TO DCNE
                     ENC;
                   NEXT(STARTPCINT):=NEXT(ECGE);
                   PATH(2):=EDGE;
                   POINT:= HEAC(ECGE);
                   PATHPT:=2;
                   IF OLC(POINT)THEN GO TO PATHFOUND;
                   IF FORWARD(PCINT)¬=0 THEN
                     BEGIN
                       COMMENT *************************************
                       *      USE PREVICUSLY FOUND INFORMATICN TO
                       *      BUILD A PATH.  FORWARD,BACK,LOWPCINT
                       *      DESCRIBE TREES INVESTIGATED USING DEPTH-
                       *      FIRST SEARCH.
                       *************************************;
                       IF PATHCCDE(STARTPOINT)>PATHCCDE(PCINT) THEN
                            GO TO NEXTBACK;
NEXTMARK:              IF PATHCODE(STARTPOINT)>LOWPOINT(POINT)
                            THEN
                          BEGIN
NEXTFCRWARD:                EDGE:= FCRWARD(POINT);
                            PCINT:=HEAD(EDGE);
                            PATHPT:= PATHPT+1;
                            PATH(PATHPT):=EDGE;
                            IF CLD(PCINT) THEN GO TO PATHFCUND;
                            IF PATHCODE(STARTPOINT)>PATHCODE(POINT)
                                 THEN GC TO NEXTBACK;
                            GO TO NEXTFORWARD
                          ENC;
                       EDGE:= BACK(POINT);
                       POINT:= HEAD(ECGE);
                       PATHPT:=PATHPT+1;
                       PATH(PATHPT):=EDGE:
                       IF CLC(PCINT) THEN GO TO PATHFOUND ELSE GO TO
                            NEXTMARK ;
NEXTBACK:              EDGE:=BACK(POINT);
                       PCINT:=HEAC(EDGE);
                       PATHPT:=PATHPT+1;
                       PATH(PATHPT):=EDGE;
                       IF CLD(PCINT) THEN GO TO PATHFOUND ELSE
                            GO TO NEXTBACK
                     END:
                   COMMENT *********************************************
                   *      USE DEPTH-FIRST SEARCH TO FIND A PATH.  SAVE
                   *      INFORMATION DESCRIBING SEARCH TREE.
                   *********************************************;
```

```
h EXTPO INT:  CODEVALUE:= CODEVALUE+1;
              PATHCCDE(P C INT):= CODEVALUE;
NEXTEDGE:     EDGE:= NCDE(POINT);
              WHILE EDGE=0 CO
                  BEGIN'
                      BACK(POINT):=LINK(PATH(PATHPT));
                      PASTPOINT:= HEAD(BACK(POINT));
                      IF(FCRWARD(PASTPCINT)=0)O  R(LOWPOINT
                              (PCINT)<LOWPOINT(PASTPOINT)) THEN
                          BEGIN
                              FORWARD(PASTPOINT):= PATH(PATHPT);
                              LCWPOINT(PASTPOINT):=LOWPOINT(POINT)
                          END;
                      POINT:= PASTPOINT;
                      PATHPT:= PATHPT-1;
                      EDGE:= NODE(POINT)
                  END;
              NODE(POINT) := NEXT(EDGE);
              V2:= HEAD(EDGE);
              I F PATHCODE(V2)=0 THEN
                  BEGIN
                      POINT:= V2;
                      PATHPT:= PATHPT+1;
                      PATH(PATHPT):=EDGE;
                      GOT ONEXTPOINT
                  END:
              I F OLD(V2)AND(V2¬=STARTPOINT)T H E N
                  BEGIN
                      PATHPT:= PATHPT+1;
                      PATH(PATHPT):= EDGE;
                      GC TO PATHFOUND
                  END;
              IF(FORWARD(PCINT)=0)OR(PATHCODE(V2)<LOWPOINT
                      (PCINT) ) THEN
                  HEGIN
                      FORWARD(PCINT):= EDGE;
                      LOWPOINT(POINT):=PATHCODE(V2)
                  END;
              G OTONEXTEDGE;
              COMMENT ***********************************************
              *       PATH FOUND.  CONVERT STACK OF EDGES TO LIST
              *       OF POINTS IN PATH.  MARK ALL EDGES AND
              *       POINTS IN PATH.
              *****************************************************;
PATHFOUND:    FOR I := 2 UNTIL PATHPT DO
                  BEGIN
                      EDGE:= PATH(I);
                      PCINT:= HEAD(EDGE);
                      FORWARD(POINT):= BACK(POINT):=0;
                      OLD(PCINT):= TRUE;
                      MARK(LINK(EDGE)):=MARK(EDGE):=TRUE;
                      PATH(I):= POINT
                  END;
DONE:       END;
```

References

[1] Fisher, G. J., "Computer recognition and extraction of planar graphs from the incidence matrix," *IEEE Transactions in Circuit Theory* CT-13, June 1966, pp. 154-163.

[2] Harary, F., *Graph Theory,* Addison-Wesley Publishing Company, Reading, Massachusetts, 1969.

[3] Holt, R., and E. Reingold, "On the time required to detect cycles and connectivity in directed graphs," Computer Science TR70-33, ✓ Cornell University, Ithaca, New York.

[4] Hopcroft, J., and R. Tarjan, "Planarity testing in v log v steps, extended abstract," Stanford University CS 201, March 1971.

[5] Lempel, A., S. Even, and I. Cederbaum, "An algorithm for planarity testing of graphs," Theory of Graphs: International Symposium: Rome, July 1966. P. Rosenstiehl, Ed., New York: Gordon and Breach, 1967, pp. 215-232.

[6] Shirey, R. W., "Implementation and analysis of efficient graph planarity testing," Ph.D. dissertation, Computer Science Department, University of Wisconsin, June 1969.