4. Variations on Threaded Code

The previous example assumed a stack was used as the basic discipline for data. Actually this assumption is unnecessary. The threaded code service routines can pass or receive data according to any convention; they may even be passed parameters if desired. The parameters of a routine can immediately follow the threaded link to the routine. As each is used by the service routine, the link pointer can be incremented to step through the parameters. For example, on the PDP-11 a two-parameter routine to copy a word *A* to a word *B* could look like this:

```
CALL:  COPY                          }
         A                           } threaded code
         B                           }
         ⋮                           }
COPY:  MOV @ (R) +, @ (R) +          } service routine
       JMP @ (R) +                   }
```

We have presented the concept of threaded code in its most basic form. There are numerous time and space optimizations which could be made. For example, it can easily be determined whether a given service routine *R* is always followed by the same other service routine *S*. If so, then *R* can end with a jump directly to *S*, leaving one less link to thread. Moreover in many cases the routine for *R* can be placed immediately before the routine for *S*, thereby eliminating the need for any jump at all. This clearly saves both space and time.

In a practical application it may be expedient to write some sections in threaded code and some in hard code, provided that shifting between modes is rapid.

5. Conclusions

We have shown that under certain circumstances threaded code provides an attractive alternative to hard code, saving space at little cost in time.

Acknowledgments. The FORTRAN IV compiler for DEC's PDP-11 has been written to generate threaded code. In the course of that project many improvements have been suggested by those associated with it. Of particular value to the author have been the ideas of Ronald Brender, David Knight, Louis Cohen, Nick Pappas, and Hank Spencer.

Received June 1971; revised December 1972

Algorithm 447

Efficient Algorithms for Graph Manipulation [H]

John Hopcroft and Robert Tarjan [Recd. 24 March 1971 and 27 Sept. 1971]
Cornell University, Ithaca, NY 14850

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.

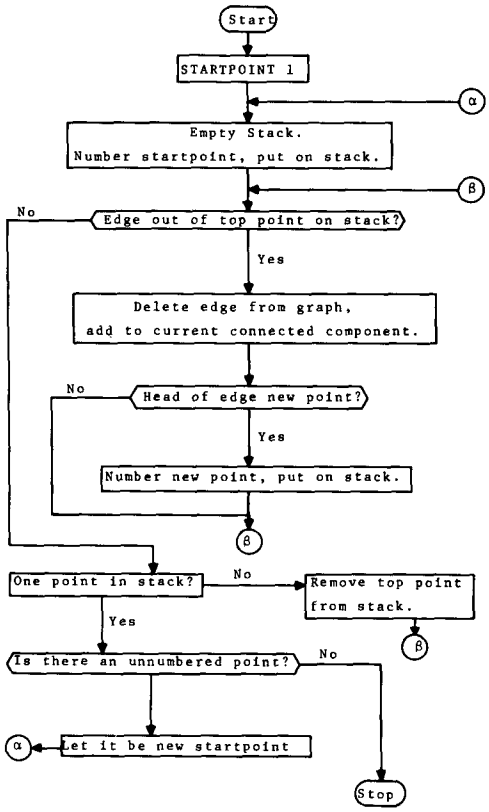
Key Words and Phrases: graphs, analysis of algorithms, graph manipulation
CR Categories: 5.32
Language: Algol

Description

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

This research was carried out while the authors were at Stanford University and was supported by the Hertz Foundation and by the Office of Naval Research under grant number N-00014-67-A-0112-0057 NR-44-402. Reproduction in whole or in part is permitted for any purpose of the United States Government.

Fig. 1. Flowchart for connected components algorithm.



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 [7]. The description of an algorithm similar to the biconnected components algorithm has just appeared [6]. 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 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 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, $E \leq 3V - 3$.) 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.

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 exhausted. 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 nondeterministic 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 nondeterministic 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 correcly 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.

Let E_1 be the set of removed edges, let E_2 be the set of edges still on the stack, and let E_3 be the set of remaining edges of *G*. Let G_1 be the subgraph of *G* made up of the edges from E_1, and let $G_2 = G - G_1$. Since G_1 and G_2 each have at most *E*-1 edges, the induction hypothesis implies that the algorithm operates correctly on both G_1 and G_2.

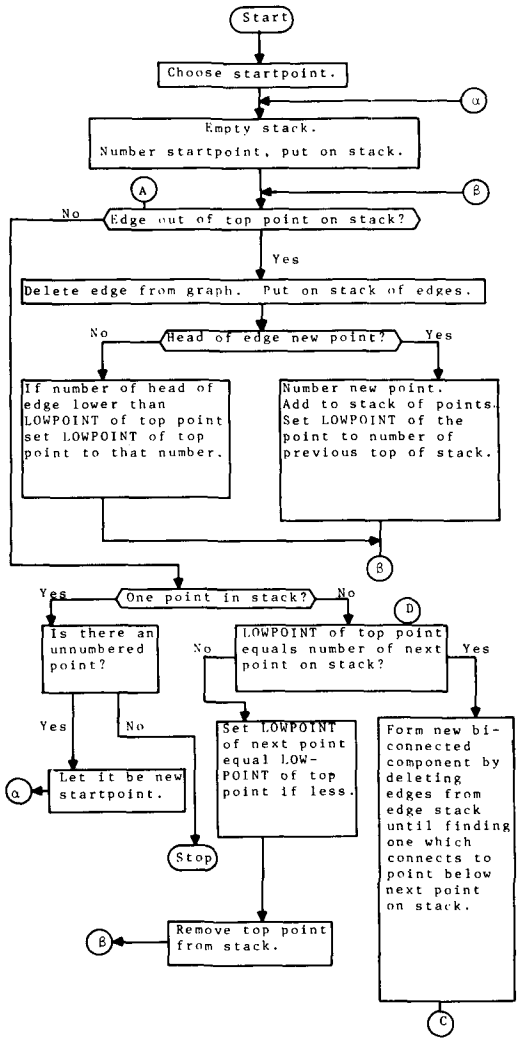
Assume that the edges for each vertex in G_1 and G_2 are listed in the same order as for *G*. Consider the sequence of steps taken when the algorithm is applied to *G*. The sequence of steps taken on G_2 can be divided into an initial sequence of steps which results in placing the edges E_1 on the stack, followed by the remaining sequence S_2. The sequence of steps taken on *G* consists of the sequence S_1, followed by the steps taken on G_2 with *p* as the start point, followed by S_2.

The behavior of the algorithm on *G* is simply the composite of its behavior on G_1 and G_2; 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 *p* and which does not contain the start point of *G*. Let the edges making up this component be subgraph G_1 of *G*; let the remainder of *G* be G_2. The algorithm operates correctly on G_1 and on G_2 by assumption. The behavior of the algorithm on *G* is a composite of its behavior on G_1 and on G_2. Assume that the edges for each vertex in G_1 and G_2 are listed in the same order as for *G*. The sequence of steps on *G* is identical to the sequence of steps on G_1 until an edge of G_2 out of vertex *p* is selected. Then the sequence of steps of *G* is identical to the sequence on G_2 with start point *p*. The remaining steps on *G* are the same as the remaining steps on G_1. But the algorithm reaches block *C* once while processing G_1 and at least once while processing G_2. 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.

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.

Fig. 2. Flowchart for biconnected components algorithm.



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 edges connecting some vertices 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 (Figures 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

Fig. 3. Flowchart for pathfinding algorithm (I).

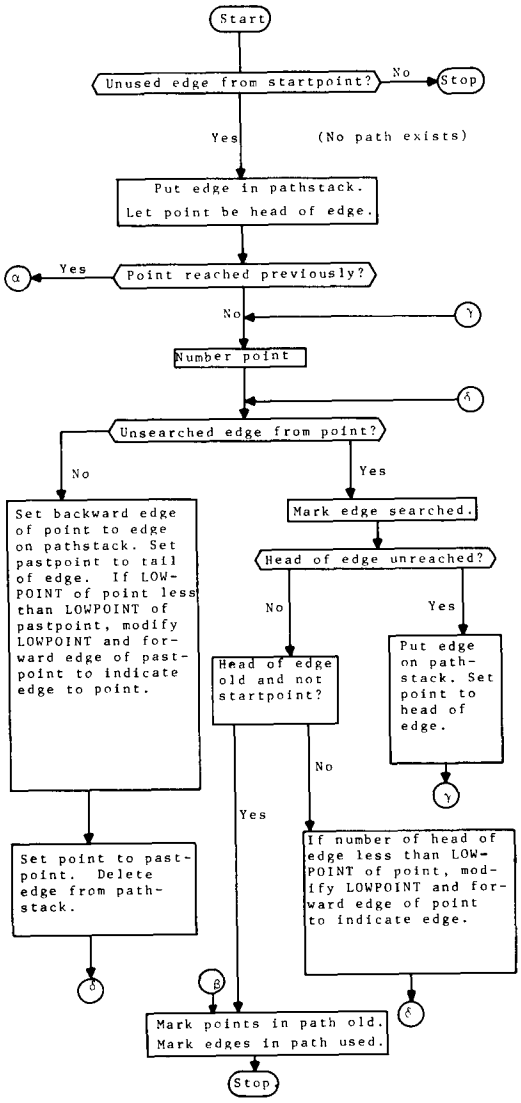
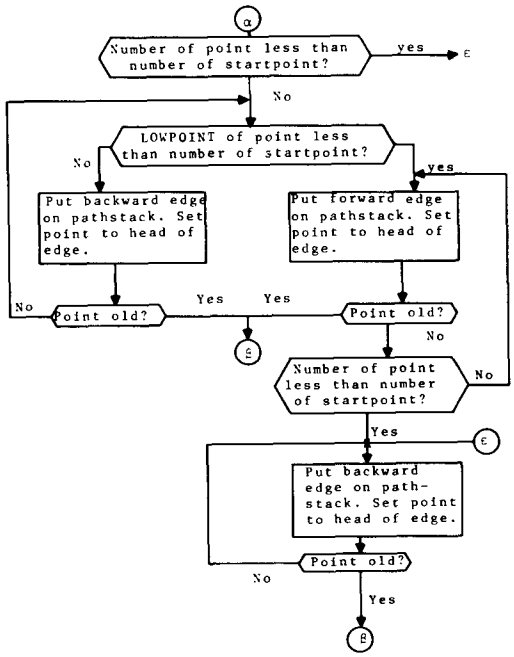


Fig. 4. Flowchart for pathfinding algorithm (II).



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 jointed 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 startpoint is 1, the endpoint is V, and the numbering satisfies the conditions above. Take edge (1, V) as the first path. Given an arbitrary startpoint, find an edge to a higher numbered vertex. Continue to find edges to successively higher numbered vertices until an old vertex is reached.

This algorithm is clearly correct and looks conceptually simple. 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.

Implementation. The algorithms for finding connected components, biconnected components, and simple paths were originally implemented and tested in Algol W. The programs were then translated to Algol for publication and tested using the OS/360 Algol compiler. 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*,*COMPONENTS*): 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.

The procedure *PATHFINDER* requires that the structural representation of the graph be stored as follows. Each edge is treated as a pair of directed edges each of which is represented by an integer between $v + 1$ and $v + 2 \times e$. If $i_1, i_2, \ldots, i_k$ are the integers corresponding to the edges out of vertex i, then initialize $NEXT(i)$ to i_1, $NEXT(i_j)$ to i_{j+1}, $1 \leq j < k$, and $NEXT(k)$ to 0. If the edge i_j terminates at vertex l, initialize $HEAD(i_j)$ to l. $LINK(i_j)$ is the integer corresponding to the edge in the other direction. For $1 \leq i \leq v$, $BACK(i)$, $FORWARD(i)$, $PATHOCDE(i)$ are initialized to 0, $LOWPOINT(i)$ is initialized to $v + 1$, $NODE(i)$ is initialized to $NEXT(i)$ and $OLD(i)$ is initialized to *FALSE*. For $v + 1 \leq i \leq v + 2 \times e$ $MARK(i)$ is initialized to *FALSE*. Before the first call of *PATHFINDER* some nonnull set of vertices must be marked as *OLD* and assigned successive *PATHCODE* values. *CODEVALUE* is set equal to the number of vertices marked as *OLD*. If this is not done the first path cannot end at an *OLD* vertex.

Further comments may be found in the program listings below.

References

1. Fisher, G.J. Computer recognition and extraction of planar graphs from the incidence matrix. *IEEE Trans. in Circuit Theory CT-13*, (June 1966), 154–163.
2. Harary, F. *Graph Theory*. Addison-Wesley, Reading, Mass., 1969.
3. Holt, R., and Reingold, E. On the time required to detect cycles and connectivity in directed graphs. Comput. Sci. TR 70–33, Cornell U. Ithaca, N.Y.
4. Hopcroft, J., and Tarjan, R. Planarity testing in v log v steps, extended abstract. Stanford U. CS 201, Mar. 1971.
5. Lempel, A., Even, S., and Cederbaum, I. An algorithm for planarity testing of graphs. Theory of Graphs: International Symposium: Rome, July 1966. P. Rosenstiehl (Ed.) Gordon and Breach, New York, 1967, pp. 215–232.
6. Paton, K. An algorithm for the blocks and cutnodes of a graph. *Comm. ACM 14*, 7(July 1971), 428–475.
7. Shirey, R.W. Implementation and analysis of efficient graph planarity testing. Ph.D. diss., Comput. Sci. Dep., U. of Wisconsin, Madison, Wis., 1969.

Algorithm

```
procedure add2 (a, b, stack, ptr);
  value a, b; integer a, b, ptr; integer array stack;
comment Procedure adds values a and b to stack stack and increases stack pointer ptr by 2;
begin
  ptr := ptr + 2; stack[ptr − 1] := a; stack[ptr] := b
end of add2;
procedure nextlink (point, val);
  value point, val; integer point, val;
comment Procedure adds directed edge (point, val) to structural representation of a graph. Global variables are described as follows. head[v+1:v+2×e] and next[1:v+2×e] contain the structural representation of the graph. freenext is the current last entry in next array;
begin
  freenext := freenext + 1; next[freenext] := next[point];
  next[point] := freenext; head[freenext] := val
end of nextlink;
integer procedure min(a, b);
  value a, b; integer a, b;
comment Procedure computes the minimum of two integers;
if a < b then min := a else min := b;
procedure connect (v, e, cptr, edgelist, components);
  value v, e; integer v, e, cptr;
  integer array edgelist, components;
comment Procedure finds the connected components of a graph. The parameters are described as follows. v and e are the number of vertices and edges of the graph. edgelist[1:2×e] is the initial list of edges of the graph. components[1:3×e] is the list of edges for each component. The list of edges for each component is preceded by an entry giving the number of edges of the component. cptr is a pointer to the last entry in components. The global variables are described as follows. head[v+1:v+2×e] and next[1:v+2×e] contain the structural representation of the graph. freenext is the last entry in the array next. The local variables are described as follows. number[1:v+1] is used for numbering the vertices during the depth first search. code contains the current highest vertex number. point is the current vertex being examined during the search. v2 is the next vertex to be examined during the search. oldptr contains the position in components to place the value of the next component. The global procedures are add2 and nextlink. A recursive depth-first search procedure is used to examine connected components of the graph;
begin
  integer array number [1:v+1];
  integer code, point, v2, oldptr, i;
  procedure connector (point, oldpt);
    value point, oldpt; integer point, oldpt;
```

```
  comment This recursive procedure finds a connected component
    using a depth-first search. The parameters are described as fol-
    lows. point is the startpoint of search. oldpt is the previous
    startpoint. Global variables are the same as for connect. The
    global procedures are add2;
  comment Examine each edge out of point;
  for i = i while next[ point] > 0 do
  begin
    comment v2 is head of edge. Delete edge from structural repre-
      sentation;
    v2 := head[next[point]];
    next[point] := next[next[point]];
    comment Has this edge been searched in the other direction?
      If so, look for another edge;
    if (number[v2] < number[point]) ∧ (v2≠oldpt) then
    begin
      comment Add edge to components;
      add2(point, v2, components, cptr);
      comment Determine if a new point has 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 step 1 until v do next[i] := 0;
  for i := 1 step 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 step 1 until v + 1 do number[i] := 0;
  for i := i while point ≤ v do
  begin
    comment Each execution of connector searches a connected
      component. After each search, find an unnumbered vertex
      and search again. Repeat until all vertices are investigated;
    number[point] := code := 1;
    oldptr := cptr := cptr + 1;
    connector(point,0);
    comment Compute number of edges of components;
      components[oldptr] := (cptr-oldptr)÷2;
    for i := i while number[point] ≠ 0 do point := point + 1
  end
end;
procedure biconnect(v, e, bptr, edgelist, bicomponents);
  value v, e; integer v, e, bptr;
  integer array edgelist, bicomponents;
begin
  comment Procedure finds biconnected components of a graph.
    The parameters are described as follows. v and e are the num-
    ber of vertices and edges of the graph. edgelist[1:2×e] is the
    initial list of edges of the graph. bicomponents[1:3×e] is the list
    of edges for each component found. Each component is pre-
    ceded by an entry giving the number of edges of the com-
    ponent. bptr is a pointer to the last entry of bicomponents. The
    global variables are described as follows. head[v+1:v+2×e]
    and next[1:v+2×e] contain the structural representation of the
    graph. freenext is the last entry in the array next. The local
    variables are described as follows. number[1:v+1] is an array
    used for numbering the vertices during the depth-first search.
    code is the current highest vertex number. edgestack[1:2×e]
    is used for storage of edges examined during search. eptr is
    a pointer to last entry in edgestack. point is the current point
    being examined during search. v2 is the next point to be ex-
    amined during search. newlowpt is the lowpoint for the bi-
    connected part of graph above and including v2. oldptr is
    pointer to position in bicomponents to place a value of next
    component. The global procedures are min, add2, and next-
    link. A recursive depth-first search procedure is used to divide
    the graph. The lowest point reachable from the current point
    without going through previously searched points is calculated.
    This information allows determination of the articulation
    points and division of the graph;
  integer array number[1:v+1], edgestack[1:2×e];
  integer code, eptr, point, v2, newlowpt, oldptr, i;
  procedure biconnector (point, oldpt, lowpoint);
    integer point, oldpt, lowpoint;
  comment Recursive procedure to search a connected component
    and find its biconnected components using depth-first search.
    The parameters are described as follows. point is the startpoint
    of the search. oldpt is the previous startpoint. lowpoint is the
    lowest point reachable on a path found during search. The
    global variables are the same as for biconnect. The global
    procedures are min and add2;
  comment Examine each edge out of point;
  for i := i while next[point] > 0 do
  begin
    comment v2 is the head of the edge. Delete edge from structural
      representation;
    integer v2;
    v2 := head[next[point]];
    next[point] := next[next[point]];
    comment If the edge has been searched in the other direction,
      then look for another edge;
    if (number[v2]<number[point]) ∧ (v2≠oldpt) then
    begin
      comment Add edge to edgestack;
      add2 (point, v2, edgestack, eptr);
      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;
        newlowpt := v + 1;
        biconnector (v2, point, newlowpt);
        comment Note that although the global variable v2 is
          changed, its value is restored upon exit from this pro-
          cedure. Recalculate lowpoint;
        lowpoint := min(lowpoint, newlowpt);
        if newlowpt ≥ number[point] then
        begin
          comment point is an articulation point. Output edges
            of component from edgestack;
          oldptr := bptr := bptr + 1;
          for i := i while number[edgestack[eptr−1]] >
            number[point] do
          begin
            add2(edgestack[eptr−1], edgestack[eptr], bicom-
              ponents, bptr);
            eptr := eptr − 2
          end;
          comment Add last edge;
          add2(point, v2, bicomponents, bptr);
          eptr := eptr − 2;
          comment Compute number of edges of component;
          bicomponents[oldptr] := (bptr-oldptr) ÷ 2
        end
      end
      else
      begin
```

```
        comment New point not found. Recalculate lowpoint;
        lowpoint := min(lowpoint, number[v2])
      end
    end
  end;
  comment Construct the structural representation of the graph;
  freenext := v;
  for i := 1 step 1 until v do next [i] := 0;
  for i := 1 step 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;
  eptr := 0; bptr := 0; point := 1; v2 := 0;
  for i := 1 step 1 until v + 1 do number[i] := 0;
  for i := i while point ≤ v do
  begin
    comment Each execution of biconnector searches a connected
      component of the graph. After each search, find an unnum-
      bered vertex and search again. Repeat until all vertices are
      examined;
    number[point] := code := 1; newlowpt := v + 1;
    biconnector(point, v2, newlowpt);
    for i := i while number[point] ¬ ≠ 0 do point := point + 1
  end
end;
procedure pathfinder (startpoint, pathpt, codevalue, path);
  integer startpoint, pathpt, codevalue;
  integer array path;
begin
  comment Procedure finds disjoint paths with arbitrary starting
    points in a biconnected graph. The points of each path are
    listed in the array path. The following variables are assumed
    global. next[1:v+2×e], head[v+1:v+2×e] and link
    [v+1:v+2×e] define the graph using singly linked edge
    lists and a set of cross reference pointers. old[1:v] and mark
    [v+1:v+2×e] indicate used points and edges. pathcode[1:v]
    is the consecutive numbering of the points. lowpoint[1:v],
    forward[1:v] and back[1:v] give information saved from depth-
    first search, node[1:v] gives the next unsearched edge from each
    point;
  integer point, pastedge, edge, pastpoint, v2, i;
  path[1] := startpoint;
  comment Choose initial edge;
  edge := next[startpoint];
  for i := i while (if edge=0 then false else mark[edge])
    do edge := next[edge];
  begin
    comment No unused edge and thus no path exists:
    next[start point] := 0; pathpt := 0:
    go to done
  end;
  next[startpoint] := next[edge]; path[2] := edge;
  point := head[edge]; pathpt := 2;
  if old[point] then go to pathfound;
  if forward[point]≠ 0 then
  begin
    comment Use previously found information to build a path.
      forward, back, lowpoint describe trees investigated using
      depth-first search;
    if pathcode[startpoint] > pathcode[point] then
      go to nextback;
nextmark:
    if pathcode[startpoint] > lowpoint[point] then
    begin
nextforward:
      edge := forward[point]; point := head[edge];
      pathpt := pathpt +1; path[pathpt]: = edge;
      if old[point] then go to pathfound;
      if pathcode[startpoint] > pathcode[point]
        then go to nextback;
      go to nextforward
    end;
    edge := back[point]; point := head[edge];
    pathpt := pathpt + 1; path[pathpt] := edge;
    if old[point] then go to pathfound else
      go to nextmark;
nextback:
    edge := back[point]; point := head[edge];
    pathpt := pathpt + 1; path[pathpt] := edge;
    if old[point] then go to pathfound else
      go to nextback
  end;
  comment Use depth-first search to find a path. Save information
    describing search tree;
nextpoint:
  codevalue := codevalue + 1; pathcode[point] := codevalue;
nextedge:
  edge := node[point];
  for i := i while edge = 0 do
  begin
    back[point] := link[path[pathpt]];
    pastpoint := head[back[point]];
    if (forward[pastpoint] = 0) ∨
      (lowpoint[point] < lowpoint[pastpoint]) then
    begin
      forward[pastpoint] := path[pathpt];
      lowpoint[pastpoint] := lowpoint[point]
    end;
    point := pastpoint; pathpt := pathpt − 1; edge := node[point]
  end;
  node[point] := next[edge]; v2 := head[edge];
  if pathcode[v2] = 0 then
  begin
    point := v2; pathpt := pathpt + 1;
    path[pathpt] := edge; go to nextpoint
  end;
  if old[v2] ∧ (v2≠startpoint) then
  begin
    pathpt := pathpt + 1; path[pathpt] := edge;
    go to pathfound
  end;
  if (forward[point]=0) ∨ (pathcode[v2] < lowpoint[point]) then
  begin
    forward[point] := edge; lowpoint[point] := pathcode[v2]
  end;
  go to nextedge;
  comment Path found. Convert stack of edges to list of points in
    path. Mark all edges and points in path;
pathfound:
  for i := 2 step 1 until pathpt do
  begin
    edge := path [i]; point := head[edge];
    forward[point] := back[point]: = 0; old[point] := true;
    mark[link[edge]] := mark [edge] := true;
    path [i] := point
  end;
done:
end
```