DIVIDING A GRAPH INTO TRICONNECTED COMPONENTS†

J.E. Hopcroft and R.E. Tarjan

TR 74-197

(new revision of TR 72-140)††

February 1974

Computer Science Department
Cornell University
Ithaca, N.Y. 14850

†This research was supported in part by the Hertz Foundation and the office of Naval Research under grant N00014-67-A-0077-0021.

††This report is a revision of "Finding the Triconnected Components of a Graph", August 1972.

DIVIDING A GRAPH INTO TRICONNECTED COMPONENTS

J.E. Hopcroft and R.E. Tarjan†
Computer Science Department
Cornell University
Ithaca, New York 14850

Abstract:

An algorithm for dividing a graph into triconnected components is presented. When implemented on a random access computer, the algorithm requires O(V+E) time and space to analyze a graph with V vertices and E edges. The algorithm is both theoretically optimal to within a constant factor and efficient in practice.

Keywords: articulation point, connectivity, depth-first search, graph, separability, separation, triconnectivity.

---

†This research was supported in part by the Hertz Foundation and the office of Naval Research under grant N00014-67-A-0077-0021.

Introduction

The connectivity properties of graphs form an important part of graph theory. Efficient algorithms for determining some of these properties are both theoretically interesting and useful in a variety of applications. This paper considers the problem of dividing a graph into triconnected components. An algorithm for this purpose is useful for analyzing electrical circuits [1], for determining whether a graph is planar [2], and for determining whether two planar graphs are isomorphic [3]. An algorithm for planarity may be used in the design of printed circuit boards; an algorithm for isomorphism of planar graphs may be used to test structural isomorphism of chemical compounds [4].

One technique which has been used to solve connectivity problems is depth-first search. In [5,6] depth-first search is applied to give efficient algorithms for determining the biconnected components of an undirected graph and for determining the strongly connected components of a directed graph. The method has also been used in an efficient algorithm for planarity testing [7,8] and in an algorithm to find dominators in a flow graph [9]. This paper applies depth-first search to the problem of finding the triconnected components of a graph. Old methods for determining these components require $O(V^3)$ steps or more, if the graph has V vertices [1,10]. The algorithm described here requires substantially less time, and it may be shown to be optimal to within a constant factor assuming a suitable model of computation.

Four sections comprise the paper. The first section presents the necessary definitions and lemmas from graph theory, and it describes depth-first search. The second section intuitively explains the triconnectivity algorithm. The third section describes preliminary calculations and a simple test to find the separation pairs of a graph. The last section gives the heart of the triconnected components algorithm, including proofs of its correctness and the derivation of time and space bounds.

In deriving time bounds on algorithms we assume a random-access computer model. A formal definition of such a model may be found in [11]. Intuitively, any logical, arithmetic, or control operation requires one step; all numbers must be integers whose absolute value is O(V), if the problem graph has V vertices. (We use the following notation for specifying bounds: if f and g are functions of x, we say "f(x) is O(g(x)) if, for some constants k_1 and k_2, $|f(x)| \leq k_1 |g(x)| + k_2$ for all x.)

Graphs, Connectivity, and Depth-First Search

The definitions used in this paper are more-or-less standard; see [12,13]. Triconnected components may be defined in several ways, all more-or-less equivalent. The results below which we give without proof follow from those of Saunders Maclaine [14]; our definitions are modified somewhat to make them more suitable for computer applications. Tutte [15] has also developed a theory of triconnected components; his definitions are equivalent to ours and to Maclaine's. The theory is also a special case of the more general theory of decomposing "clutters" into "chunks" due to Edmonds and Cunningham [16].

A *graph* $G = (\mathcal{V},\mathcal{E})$ consists of a set $\mathcal{V}$ containing V *vertices* and a set $\mathcal{E}$ containing E *edges*. If the edges are ordered pairs(v,w) of distinct vertices, the graph is *directed*; v is called the *tail* and w the *head* of the edge. If the edges are unordered pairs of distinct vertices, also denoted by (v,w), the graph is *undirected*. If $\mathcal{E}$ is a multiset; that is, any edge may occur several times, then G is a *multigraph*. If (v,w) is an edge of a multigraph G, vertices v and w are *adjacent*. Edge (v,w) is *incident* to vertices v and w; v and w are *incident* to (v,w). If $\mathcal{E}'$ is a set of edges in G, $\mathcal{V}(\mathcal{E}')$ is the set of vertices incident to one or more of the edges in $\mathcal{E}'$. If S is a set of vertices in G, $\mathcal{E}(S)$ is the set of edges incident to at least one vertex in S.

If G is a multigraph, a path p: $v \overset{*}{\Rightarrow} w$ in G is a sequence of vertices and edges leading from v to w. A path is simple if all its vertices are distinct. A path p:$v \overset{*}{\Rightarrow} v$ is a cycle if all its edges are distinct and the only vertex to occur twice on p is v, which occurs exactly twice. Two cycles which are cyclic permutations of each other are considered to be the same cycle. The undirected version of a directed multigraph is the multigraph formed by converting each edge of the directed multigraph into an undirected edge. An undirected multigraph is connected if every pair of vertices v and w in G is connected by a path. If $G=(\mathcal{V},\mathcal{E})$ and $G'=(\mathcal{V}',\mathcal{E}')$ are two multigraphs such that $\mathcal{V}' \subseteq \mathcal{V}$ and $\mathcal{E}' \subseteq \mathcal{E}$, then G' is a subgraph of G. A multigraph having exactly two vertices v,w and one or more edges (v,w) is called a bond.

A (directed, rooted) tree T is a directed graph whose undirected version is connected, having one vertex which is the head of no edges (called the root), and such that all vertices except the root are the head of exactly one edge. The relation " (v,w) is an edge of T " is denoted by $v \rightarrow w$. The relation "there is a path from v to w in T" is denoted by $v \overset{*}{\rightarrow} w$. If $v \rightarrow w$, v is the father of w and w is a son of v. If $v \overset{*}{\rightarrow} w$, v is an ancestor of w and w is a descendant of v. The set of descendants of a vertex v is denoted by D(v). Every vertex is an ancestor and a descendant of itself. If G is a directed multigraph, a tree T is a spanning tree of G if T is a subgraph of G and T contains all the vertices of G.

Let P be a directed multigraph, consisting of two disjoint sets of edges, denoted by $v \to w$ and $v \dashrightarrow w$. Suppose P satisfies the following properties:

(i) The subgraph T containing the edges $v \to w$ is a spanning tree of P.

(ii) If $v \dashrightarrow w$, then $w \xrightarrow{*} v$. That is, each edge not in the spanning tree T of P connects a vertex with one of its ancestors in T.

Then P is called a <u>palm tree</u>. The edges $v \dashrightarrow w$ are called the <u>fronds</u> of P.

A connected multigraph G is <u>biconnected</u> if for each triple of distinct vertices v,w and a in V there is a path $p: v \overset{*}{\Rightarrow} w$ such that a is not on the path p. If there is a distinct triple v,w,a such that a is on every path $p: v \overset{*}{\Rightarrow} w$, then a is called a <u>separation point</u> (or an <u>articulation point</u>) of G. We may partition the edges of G so that two edges are in the same block of the partition if and only if they belong to a common cycle. Let $G_i = (V_i, E_i)$ where E_i is the set of edges in the i^{th} block of the partition and $V_i = V(E_i)$. Then:

(i) Each G_i is biconnected.

(ii) No G_i is a proper subgraph of a biconnected subgraph of G.

(iii) Each vertex of G which is not an articulation point of G occurs exactly once among the V_i and each articulation point occurs at least twice.

(iv) For each $i, j, i \neq j$, $V_i \cap V_j$ contains at most one vertex; furthermore, this vertex (if any) is an articulation point.

The subgraphs G_i of G are called the biconnected components of G. The biconnected components of G are unique.

Let {a,b} be a pair of vertices in a biconnected multigraph G. Suppose the edges of G are divided into equivalence classes $E_1, E_2, \ldots, E_n$ such that two edges which lie on a common path not containing any vertex of {a,b} except as an end point are in the same class. The classes E_i are called the separation classes of G with respect to {a,b}. If there are at least two separation classes, then {a,b}is a separation pair of G unless (1) there are exactly two separation classes and one class consists of a single edge or (2) there are exactly three classes each consisting of a single edge.

If G is a biconnected multigraph such that no pair {a,b } is a separation pair of G, then G is triconnected. Let {a,b } be a separation pair of G. Let the separation classes of G with respect to {a,b} be $E_1, E_2, \ldots, E_n$. Let $E' = \bigcup_{i=1}^{k} E_i$ and $E'' = \bigcup_{i=k+1}^{n} E_i$ be such that $|E'| \geq 2$, $|E''| \geq 2$. Let $G_1 = (V(E'), E' \cup \{(a,b)\})$, $G_2 = (V(E''), E'' \cup \{(a,b)\})$. The graphs G_1 and G_2 are called split graphs of G with respect to {a,b}. Replacing a multigraph G by two split graphs is called splitting G. There may be many possible ways to split a graph, even with respect to a fixed separation pair {a,b}. A splitting operation is denoted by s(a,b,i); i is a label distinguishing this split operation from other splits. The new edges (a,b) added to G_1 and G_2 are called virtual edges; they are labelled to identify them with the split. A virtual edge (a,b) associated with split s(a,b,i) will be denoted by (a,b,i). If G is biconnected, then any split graph of G is also biconnected.

Suppose a multigraph G is split, the split graphs are split, and so on, until no more splits are possible (each graph remaining is triconnected). The graphs constructed in this way are called the split components of G. The split components of a multigraph are not necessarily unique.

Lemma 1: Let G = (V,E) be a multigraph with $|E| \geq 3$. Let $G_1, G_2, \ldots, G_m$ be the split components of G. Then the total number of edges in $G_1, G_2, \ldots, G_m$ is bounded by $3|E|-6$.

Proof: The lemma is proved by induction on the number of edges of G. If G has 3 edges the lemma is immediate, because G cannot be split. Suppose the lemma is true for graphs with n-1 edges and suppose G has n edges. If G cannot be split the lemma is true for G. Suppose on the other hand that G can be split into G' and G", where G' has k+1 edges and G" has n-k+1 edges for some $2 \leq k \leq n-2$. By induction, the total number of edges in $G_1, G_2, \ldots, G_m$ must be bounded by $3(k+1) - 6 + 3(n-k+1) - 6 = 3n-6$. Thus by induction the lemma is true.

In order to get unique triconnected components we must partially reassemble the split components. Suppose $G_1=(V_1,E_1)$ and $G_2=(V_2,E_2)$ are two split components both containing a virtual edge (a,b,i). Let

$$G = (V_1 \cup V_2,\ (E_1-\{(a,b,i)\}) \cup (E_2-\{(a,b,i)\})).$$

Then G is called a merge graph of G_1 and G_2; the merge operation will be denoted by $m(a,b,i)$. Merging is the inverse of splitting; if we perform a sufficient number of merges on the split components of a multigraph we recreate the original multigraph.

The split components of a multigraph are of three types: triple bonds, of the form $(\{a,b\},\{(a,b),(a,b),(a,b)\})$; triangles, of the form $(\{a,b,c\},\{(a,b),(a,c),(b,c)\})$; and triconnected graphs. Let G be a multigraph whose split components are a set of triple bonds $\mathscr{B}_3$, a set of triangles $\mathscr{T}$, and a set of triconnected graphs $\mathscr{G}$. Suppose the triple bonds $\mathscr{B}_3$ are merged as much as possible to give a set of bonds $\mathscr{B}$, and that the triangles $\mathscr{T}$ are merged as much as possible to give a set of polygons $\mathscr{P}$. Then the set of graphs $\mathscr{B} \cup \mathscr{P} \cup \mathscr{G}$ is the set of triconnected components of G. If G is an arbitrary multigraph, the triconnected components of the biconnected components of G are called the triconnected components of G.

Lemma 2: The triconnected components of a graph G are unique.

Proof: See [14,16,17].

Figure 1 illustrates a biconnected graph G with several separation pairs. Figure 2 gives the split components of G. The triconnected components of G are formed by merging triangle (1,8,4) and triangle (4,5,8).

Graph algorithms require a systematic way of exploring a multigraph. We will use a method called depth-first search. To carry out a depth-first search of G, start from some vertex s and choose an edge leading from s to follow. Traversing the edge leads to a new vertex. Continue in this way, at each step selecting an unexplored edge leading from the most recently reached vertex which still has unexplored edges. If G is connected, each edge is traversed exactly once.

If G is undirected, a search of G imposes a direction on each edge of G given by the direction in which the edge is traversed during the search. Thus the search converts G into a directed multigraph G!

Lemma 3: Let P be the directed multigraph generated by a depth-first search of a connected undirected multigraph G. Then P is a palm tree.

<u>Proof</u>: See [5].

Depth-first search is important because the structure of paths in a palm tree is very simple. To implement a depth-first search of a multigraph, we use a simple recursive procedure which keeps a stack of the old vertices with possibly unexplored edges. To represent a multigraph, we use a set of <u>adjacency lists</u>, one for each vertex. If v is a vertex, adjacency list A(v) contains all w such that (v,w) is an edge of G. These lists together comprise an <u>adjacency structure</u> for G. If G is undirected, each edge (v,w) is represented twice, once in A(v) and once in A(w). If G is directed, each edge is represented once.

Below is a recursive procedure to carry out a depth-first search. The exact search depends upon the order of edges in the adjacency lists. The procedure numbers the vertices from 1 to V in the order they are reached during the search, in addition to identifying tree arcs and fronds. Reference [5] gives a proof that the procedure is correct and requires O(V+E) time to execute. It is easy to see that the vertices are numbered so that $\text{NUMBER}(v) < \text{NUMBER}(w)$ if $v \overset{*}{\rightarrow} w$ in the generated spanning tree.

```
begin comment Routine for depth-first search of a multigraph G
    represented by adjacency lists A(v).  Variable n denotes
    the last number assigned to a vertex;
  integer n;
  procedure DFS(v,u); begin comment vertex u is the father of
      vertex v in the spanning tree being constructed.  The
      graph to be searched is represented by a set of adja-
      cency lists A(v);
    n:=NUMBER(v):=n+1;
  a: comment dummy statement;
    for w∈A(v) do begin
      if NUMBER(w)=0 then begin
          comment w is a new vertex;
          mark (v,w) as a tree arc;
          DFS(w,v);
        b: comment dummy statement;
      end
      else if (NUMBER(w)<NUMBER(v)) and ((w≠u) or ¬FLAG(v))
        then begin
          comment the test is necessary to avoid exploring an
            edge in both directions.  FLAG(v) becomes false
            when the entry in A(v) corresponding to tree arc
            (u,v) is examined;
          mark (v,w) as a frond;
        c: comment dummy statement;
      end;
      if w=u then FLAG(v)=false;
    end;
  end;
  n:=0;
```

```
  for i:=1 until V do begin
    NUMBER(i):=0;
    FLAG(i):=true;
  end;
  comment the search starts at vertex s;
  DFS(s,0);
end;
```

The dummy statements a,b,c, will be replaced when DFS is used to calculate other information about the graph. Figure 3 depicts the palm tree formed by applying DFS to the graph in Figure 1.

An Outline of the Triconnectivity Algorithm

This section sketches the ideas behind the triconnectivity algorithm. Later sections develop the detailed components. The algorithm is based on an idea of Auslander, Parter, and Goldstein [18,19] for testing the planarity of graphs. Auslander, Parter, and Goldstein's idea gives rise to an O(V) time algorithm for testing planarity, if depth-first search is used to order the calculations [7,8]. The same idea gives an O(V+E) time algorithm for finding triconnected components.

Let G be an arbitrary biconnected multigraph. Suppose a cycle c is found in G. When the cycle is deleted from G, certain connected pieces remain, called segments. Auslander and Parter show that G is planar if and only if:

(i) Any subgraph of G consisting of c plus a single segment is planar; and,

(ii) The segments may be combined consistently to give a planar embedding of the entire graph.

An efficient planarity algorithm may be developed from this result [7,8]. A similar result holds for the separation pairs of G; namely:

Lemma 4: Let G be a biconnected multigraph and let c be a cycle in G. Let $S_1,\ldots,S_n$ be the subgraphs of G-c such that e_1 and e_2 are edges of S_i if and only if some path p in G contains both

e_1 and e_2 and no vertex of c lies between e_1 and e_2 in p. The segments S_i and the cycle c partition the edges of G. Let $\{a,b\}$ be a separation pair of G such that (a,b) is not a multiple edge. Then:

(1) Either a and b both lie on c or a and b both lie in some segment S_i.

(2) Suppose a and b both lie on c. Let p_1 and p_2 be the two paths comprising c which join a and b. Then either:

  (i) Some segment S_i with at least two edges has only a and b in common with c, and some vertex v does not lie in S_i ($\{a,b\}$ is called a <u>type 1</u> separation pair); or,

  (ii) No segment contains a vertex $v \neq a$, b in p_1 and a vertex $w \neq a,b$ in p_2, and p_1 and p_2 each contain a vertex besides a and b ($\{a,b\}$ is called a <u>type 2</u> separation pair);

(3) Conversely, any pair $\{a,b\}$ which satisfies (i) or (ii) is a separation pair.

It is easy to prove this lemma; a more technical version is proved in the next section. Lemma 4 gives rise to an efficient recursive algorithm for finding split components. We find a cycle in G and determine the segments formed when it is deleted. We test each segment for separation pairs by applying the algorithm recursively and we test the cycle for separation pairs by checking the criteria in Lemma 4. Recursive application of the algorithm requires finding cycles in subgraphs of G formed by combining a segment S_i and the initial cycle c.

We can make this algorithm very efficient by ordering the calculations using depth-first search. Each recursive call on the algorithm requires that we find a cycle in the piece of the graph to be tested for separation pairs. This cycle will consist of a simple path of edges not in previously found cycles, plus a simple path of edges in old cycles. We use depth-first search to divide the graph into simple paths which may be assembled into these cycles. The first cycle c will consist of a sequence of tree arcs followed by one frond in P, the palm tree formed from G by depth-first search. The numbering of vertices is such that the vertices are in order by number along the cycle. Each segment will consist either of a single frond (v,w) or of a tree arc(v,w) plus a subtree with root w, plus all fronds wich lead from the subtree. The search explores the segments in decreasing order of v and partitions each into simple paths consisting of a sequence of tree arcs followed by one frond.

Finding paths actually requires two searches because the pathfinding search must be carried out in a special order if it is to succeed, and certain preliminary calculations are necessary. The section on finding separation pairs describes the pathfinding process in detail and includes a version of Lemma 4 which characterizes separation pairs in terms of the generated paths. The section on finding split components indicates how these results may be used to determine the split components of a biconnected multigraph in O(V+E) time.

To determine the triconnected components of an arbitrary multigraph, we eliminate multiple edges by splitting them off, creating a set of bonds with three edges. This requires O(V+E) time if implemented correctly. Then we find the biconnected components of the resultant graph using the O(V+E) algorithm described in [5,6]. Next, the split components of each biconnected component are found using the algorithm outlined above and presented in detail in the next two sections. This gives us the split components of the entire graph. The total size of the split components is O(V+E) by Lemma 1. Next we identify the set of triple bonds $\mathscr{B}_3$ and the set of triangles $\mathscr{T}$. For each of these two sets, we construct an auxiliary graph S whose vertices are the elements of the set; two split components are joined by an edge in an auxiliary graph if they have a a common virtual edge. The connected components of $S(\mathscr{B}_3)$ and $S(\mathscr{T})$ correspond to the bonds and polygons which are triconnected components of G. Finding these bonds and polygons requires O(V+E) time. Below is an outline of the entire algorithm.

```
procedure TRICONNECTIVITY(G); begin comment an outline of the
     triconnected components algorithm;
     A:  split off multiple edges of G to form a set of triple
           bonds and a graph G';
     B:  find biconnected components of G';
         for each biconnected component C of G' do
         C:  find split components of C;
     D:  combine triple bonds and triangles into bonds and
         polygons by finding connected components of corres-
         ponding auxiliary graphs;
end;
```

Steps A,B, and D all require O(V+E) time if correctly implemented. Implementation of step B is described in [5]; implementation of steps A and D is left as an exercise. The hard step is step C, whose implementation is described in the next two sections. Based on the results of these sections, the entire triconnectivity algorithm has O(V+E) time and space bounds.

Finding Separation Pairs

Let $G = (\mathcal{V}, \mathcal{E})$ be a biconnected multigraph with V vertices and E edges. The main problem in dividing G into its split components lies in finding its separation pairs. This section gives a simple criterion, based upon depth-first search, for identifying the separation pairs of a multigraph. Two depth-first searches and some auxiliary calculations must be carried out. These calculations form the first part of the split components algorithm and are outlined below. The definitions for the quantities LOWPT1, ND, etc., used in the outline will be given subsequently.

1. Perform a depth-first search on the multigraph G, converting G into a palm tree P. Number the vertices of G in the order they are reached during the search. Calculate LOWPT1(v), LOWPT2(v), ND(v), and FATHER(v) for each vertex v in P.

2. Construct an acceptable adjacency structure A for P by ordering the edges in the adjacency structure according to the LOWPT1 and LOWPT2 values.

3. Perform a depth-first search of P using the adjacency structure A. Renumber the vertices of A from V to 1 in the order they are <u>last</u> examined during the search. Partition the edges into disjoint simple paths. Recalculate LOWPT1(v) and LOWPT2(v) using the new vertex numbers. Calculate A1(v), DEGREE(v), and HIGHPT(v) for each vertex v.

The details of these calculations appear below. From Steps 1,2, and 3 we get enough information to rapidly determine the separation pairs of G. Lemma 13 gives a condition for this purpose.

Suppose G is explored in a depth-first manner, giving a palm tree P. Let the vertices of P be numbered from 1 to V so that $v \overset{*}{\rightarrow} w$ in P implies $v<w$, if we identify vertices by their number. For any vertex v in P, let FATHER(v) be the father of v in the spanning tree of P. Let ND(v) be the number of descendants of v. Let $\mathrm{LOWPT1}(v)=\min(\{v\} \cup \{w \mid v \overset{*}{\rightarrow} {-}{\rightarrow} w\})$. That is, LOWPT1(v) is the lowest vertex reachable from v by traversing zero or more tree arcs in P followed by at most one frond. Let $\mathrm{LOWPT2}(v) = \min[\{v\} \cup (\{w \mid v \overset{*}{\rightarrow} {-}{\rightarrow} w\} - \{\mathrm{LOWPT1}(v)\})]$. That is, LOWPT2(v) is the <u>second</u> lowest vertex reachable from v by traversing zero or more tree arcs followed by at most one frond of P, unless LOWPT1(v)=v. In this case LOWPT2(v)=v.

<u>Lemma 5</u>: $\mathrm{LOWPT1}(v) \overset{*}{\rightarrow} v$ and $\mathrm{LOWPT2}(v) \overset{*}{\rightarrow} v$ in P.

<u>Proof</u>: $\mathrm{LOWPT1}(v) \le v$ by definition. If LOWPT1(v)=v the result is immediate. If LOWPT1(v)<v there is a frond $u {-}{\rightarrow} \mathrm{LOWPT1}(v)$ such that $v \overset{*}{\rightarrow} u$. Since $u {-}{\rightarrow} \mathrm{LOWPT1}(v)$ is a frond, $\mathrm{LOWPT1}(v) \overset{*}{\rightarrow} u$. Since P is a tree, $v \overset{*}{\rightarrow} u$ and $\mathrm{LOWPT1}(v) \overset{*}{\rightarrow} u$, either $v \overset{*}{\rightarrow} \mathrm{LOWPT1}(v)$ or $\mathrm{LOWPT1}(v) \overset{*}{\rightarrow} v$. But LOWPT1(v)<v. Thus it must be the case that $\mathrm{LOWPT1}(v) \overset{*}{\rightarrow} v \overset{*}{\rightarrow} u$, and the lemma holds for LOWPT1(v). The proof is the same for LOWPT2(v).

Lemma 6: Suppose LOWPT1(v) and LOWPT2(v) are defined relative to some numbering for which $v \overset{*}{\rightarrow} w$ in P implies NUMBER(v) < NUMBER(w). Then LOWPT1(v) and LOWPT2(v) identify unique vertices independent of the numbering used.

Proof: LOWPT1(v) always identifies an ancestor of vertex v. Furthermore LOWPT1(v) is the lowest numbered ancestor of v with a certain property relative to the palm tree P. Since the order of the ancestors of v corresponds to the order of their numbers, LOWPT1(v) identifies a unique vertex independent of the numbering; namely, the first ancestor of v along the path $1 \overset{*}{\rightarrow} v$ which has the desired property. (Any satisfactory numbering assigns 1 to the root of P.) The proof is the same for LOWPT2(v).

The LOWPT values of a vertex v depend only on the LOWPT values of sons of v and on the fronds leaving v; it is easy to see that if vertices are identified by number,

$$\text{LOWPT1}(v) = \min(\{v\} \cup \{\text{LOWPT1}(w) \mid v \rightarrow w\} \cup \{w \mid v \dashrightarrow w\}) \quad \text{and}$$

$$\text{LOWPT2}(v) = \min(\{v\} \cup ((\{\text{LOWPT1}(w) \mid v \rightarrow w\} \cup \{\text{LOWPT2}(w) \mid v \rightarrow w\} \cup \{w \mid v \dashrightarrow w\}) - \{\text{LOWPT1}(v)\})).$$

We also have $\text{ND}(v) = 1 + \sum_{v \rightarrow w} \text{ND}(w)$. We may calculate LOWPT values, ND, and FATHER for all vertices in O(V+E) time by inserting the following statements for the dummy statements a,b,c in DFS. Numbering the vertices in the order they are reached during the search clearly guarantees that $v \overset{*}{\rightarrow} w$ implies $v<w$.

```
comment additions to DFS for step 1;
a: LOWPT1(v):=LOWPT2(v):=NUMBER(v);
   ND(v):=1;

b: if LOWPT1(w)<LOWPT1(v) then begin
      LOWPT2(v):=min{LOWPT1(v),LOWPT2(w)};
      LOWPT1(v):=LOWPT1(w);
   end else if LOWPT1(w)=LOWPT1(v) then
          LOWPT2(v):=min{LOWPT2(v),LOWPT2(w)};
   else LOWPT2(v):=min{LOWPT2(v),LOWPT1(w)};
   ND(v):=ND(v)+ND(w);
   FATHER(w):=v;

c: if NUMBER(w)<LOWPT1(v) then begin
     LOWPT2(v):=LOWPT1(v);
     LOWPT1(v):=NUMBER(w);
   end else if NUMBER(w)>LOWPT1(v) then
     LOWPT2(v):=min{LOWPT2(v),NUMBER(w)};
```

It is easy to verify that DFS as modified above will compute LOWPT1,LOWPT2,ND, and FATHER correctly in O(V+E) time. (See [8,17].) LOWPT1 may be used to test the biconnectivity of G, as described in [5]. One related lemma is important:

Lemma 7: If G is biconnected and $v \to w$, $\mathrm{LOWPT1}(w)<v$ unless $v=1$, in which case $\mathrm{LOWPT1}(w)=v=1$. Also, $\mathrm{LOWPT1}(1)=1$.

Proof: See [5].

Let ϕ be the mapping from the edges of P into $\{1,2,\cdots,2V+1\}$ defined by:

(i) if $e = v \dashrightarrow w$, $\phi(e) = 2w+1$.

(ii) if $e = v \rightarrow w$ and $\mathrm{LOWPT2}(w) < v$, $\phi(e) = 2\mathrm{LOWPT1}(w)$.

(iii) if $e = v \rightarrow w$ and $\mathrm{LOWPT2}(w) \geq v$, $\phi(e) = 2\mathrm{LOWPT1}(w)+1$.

Let A be an adjacency structure for P. A is called <u>acceptable</u> if the edges e in each adjacency list of A are ordered according to increasing value of $\phi(e)$.

<u>Lemma 8</u>: Let P be a palm tree of a biconnected graph G whose vertices are numbered so that $v \overset{*}{\rightarrow} w$ in P implies $v < w$. Then the acceptable adjacency structures of P are independent of the exact numbering scheme.

<u>Proof</u>: If $v \rightarrow w$ in P, then by Lemma 5, LOWPT2(w) is an ancestor of w. By Lemma 6, LOWPT2(w) is a fixed vertex independent of the numbering. Since the order of the ancestors is independent of the numbering, the question as to whether LOWPT2(w) is less than v is independent of the numbering. Since G is biconnected if $v \rightarrow w$ in P, then $\mathrm{LOWPT1}(w) \leq v$ by Lemma 7. By Lemma 5, LOWPT1(w) is an ancestor of w. Since $\mathrm{LOWPT1}(w) \leq v$, LOWPT1(w) must be an ancestor of v. By Lemma 6, the vertex corresponding to LOWPT1(w) is independent of the numbering scheme. Similarly if $v \dashrightarrow w$, then by Lemma 3 and the definition of a palm tree, w is an ancestor of v. But the order of the ancestors of v is identical to the order of their numbers, and this order is independent of the numbering. Thus the acceptable adjacency structures A for P depend only on P and not on the exact numbering.

In general, a palm tree P has many acceptable adjacency structures. Given a satisfactory numbering of the vertices of P, we may easily construct an acceptable adjacency structure A by using a radix sort with 2V+1 buckets. The following procedure gives the sorting algorithm, which is Step 2 of the calculations. All vertices are identified by number. It is obvious that the sorting procedure requires O(V+E) time.

```
comment construction of ordered adjacency lists;
for i:=1 until 2*V+1 do BUCKET(i):=the empty list;
for (v,w) an edge of G do begin
  compute ϕ((v,w));
  add (v,w) to BUCKET(ϕ(v,w));
end;
for i:=1 until V do A(i):= the empty list;
for i:=1 until 2*V+1 do
  for (v,w)∈BUCKET(i) do add w to end of A(v);
```

In Step 3 of the calculations we perform a depth-first search of P using the acceptable adjacency structure A given by step 2. This search generates a set of paths in the following manner: each time we traverse an edge we add it to the path being built. Each time we traverse a frond, the frond becomes the last edge of the current path. Thus each path consists of a sequence of tree arcs followed by a single frond. Because of the ordering imposed on A, each path terminates at the lowest possible vertex, the initial path is a cycle, and each path except the first is simple and has only its initial and terminal vertex in common with previously generated paths [7,8].

If $p:s \overset{*}{\Rightarrow} f$ is a generated path, we may form a cycle by adding the tree path $f \overset{*}{\rightarrow} s$ to p. The cycles formed in this way are the cycles generated by recursive calls on the basic triconnectivity algorithm explained in the last section.

We need only minimal information about the paths. Let the vertices of P be numbered so that $v \overset{*}{\rightarrow} w$ implies $v \leq w$. Let A1(v) be the first vertex in A(v). If $v \dashrightarrow w$ is the *first* frond explored in step 3 which terminates at w, let HIGHPT(w)=v. Let DEGREE(v) be the number of edges incident to vertex v. Step 3 numbers the vertices from V to 1 in the order they are *last* examined during the search. It is clear that this numbering guarantees that $v<w$ if $v \overset{*}{\rightarrow} w$. Step 3 also computes LOWPT1(v), HIGHPT(v), A1(v), and DEGREE(v) with respect to the numbering. The following routine based on DFS will perform step 3 in O(V+E) time:

```
step 3:  begin comment routine to generate paths in a biconnected
             palm tree with specially ordered adjacency lists A(v).
             Vertex s is a global variable denoting the start ver-
             tex of the current path  s is initialized to 0.  Vari-
             able m denotes the last number assigned to a vertex;

           procedure PATHFINDER(v); begin
X:           NEWNUM(v):=m-ND(v)+1;
             for w∈A(v) do
               if s=0 then begin
                 s:=v;
                 start new path;
               end;
               add (v,w) to current path;
               if v→w then begin
                 PATHFINDER(w);
Y:               m:=m-1;
               end else begin comment v-→w;
                 if HIGHPT(NEWNUM(w))=0 then HIGHPT(NEWNUM(w)):=
                   NEWNUM(v);
                 output current path;
                 s:=0;
               end;
           end;
           s:=0;
Z:         m:=V;
           for i:=1 until V do NEWNUM(i):=HIGHPT(i):=0;
           comment vertex 1 is the start vertex of the search;
           PATHFINDER(1);
           for all vertices V do
               compute A1(v), DEGREE(v) and LOWPT1(v) and
               LOWPT2(v) using the new numbering;
         end;
```

Step 3 numbers the vertices from V to 1 in the order they are last reached during the search. However, each vertex must actually be assigned a number the first time it is reached, in order for the calculation of HIGHPT to proceed correctly. In order to accomplish this, variable i is set equal to V when the search begins (statement Z). The value of i is decreased by one each time a new vertex is discovered (statement Y). Thus when a vertex v is first reached, i is equal to the number we want to assign to v minus the number of vertices to be examined before v is examined for the last time. But the vertices to be reached between the time v is first examined and the time v is last examined are just the proper descendants of v. Thus if we assign the number i- ND(v)+1 to v when v is first examined (statement X), the numbering will be correct. The other calculations performed in Step 3 are straightforward and easy to implement. The palm tree for the graph G of Figure 1 is illustrated in Figure 4 along with LOWPT values and the set of paths generated by Step 3.

Let G be a biconnected multigraph on which Steps 1, 2, and 3 have been performed, giving a palm tree P and the sets of values defined above. Let A with adjacency lists A(v) be the acceptable adjacency structure constructed in Step 2. Let the vertices of G be identified by the numbers assigned in Step 3. We need one more definition. If u→v and v is the first entry in A(u), then v is called the first son of u. (For each vertex v, A1(v), the first son of v if one exists, is calculated in

Step 3.) If $u_0 \to u_1 \to \cdots \to u_n$ and u_i is a first son of u_{i-1} for $1 \leq i \leq n$, then u_n is called a first descendant of u_0. The sequence of tree arcs $u_0 \to u_1 \to u_2 \to \cdots \to u_n$ is part of a path generated by Step 3. The lemmas below give the properties we need to determine the separation pairs of G.

Lemma 9: Let A(u) be the adjacency list of vertex u. Let $u \to v$ and $u \to w$ be tree arcs with v occurring before w in A(u). Then $u < w < v$.

Proof: Step 3 numbers the vertices from V to 1 in the order they are last examined in the search. If $u \to v$ is explored before $u \to w$, v will be examined last before w is examined last, and v will receive a higher number. Clearly u will be last examined after both v and w are last examined, so u receives the smallest number of the three vertices.

Lemma 10: A is acceptable with respect to the numbering given by Step 3.

Proof: The sorting in Step 2 creates an acceptable adjacency structure for the original numbering. By Lemma 9, $u \to v$ implies $u < v$ and hence by Lemma 8, A is acceptable for the new numbering.

Lemma 11: If v is a vertex and D(v) is the set of descendants of v, then $D(v) = \{x \mid v \leq x < v + ND(v)\}$. If w is a first descendant of v, then $D(v) - D(w) = \{x \mid v \leq x < w\}$.

Proof: Suppose we reverse all the adjacency lists $A(v)$ and use them to specify a depth-first search of P. Vertices will be examined for the first time in ascending order from 1 to V, if vertices are identified by their Step 3 number. Thus descendants of v are assigned consecutive numbers from v to $v+ND(v)-1$. If w is a first descendant of v, vertices in $D(w)$ will be assigned numbers after all vertices in $D(v) - D(w)$. Thus $D(v) - D(w)=\{x \mid v \leq x < w\}$.

Lemma 12: Let $\{a,b\}$ be a separation pair in G with $a<b$. Then $a \xrightarrow{*} b$ in the spanning tree T of P.

Proof: Since $a<b$, a cannot be a descendant of b. Suppose b is not a descendant of a. Let E_i for $1 \leq i \leq k$ be the separation classes with respect to $\{a,b\}$. Let $S=\mathcal{V} - D(a) - D(b)$. The vertices S define a subtree in T containing neither a nor b, so $E(S)$ must be contained in some separation class, say E_1. Let c be any son of a. $E(D(c))$ must be contained in some separation class. But since G is biconnected and $a \neq 1$, $LOWPT1(c)<a$, by Lemma 7. Thus some edge is incident to a vertex in S and to a vertex in $D(c)$. Thus $E(D(c)) \subseteq E_1$. A similar argument shows that edges incident to any descendant of b are in E_1. But this means that $E_1=E$, and $\{a,b\}$ cannot be a separation pair.

Lemma 13: Suppose $a<b$. Then $\{a,b\}$ is a separation pair of G if and only if either (1), (2), or (3) below holds.

(1) There are distinct vertices $r \neq a,b$ and $s \neq a,b$ such that $b \rightarrow r$, LOWPT1$(r)=a$, LOWPT2$(r) \geq b$, and s is not a descendant of r. ($\{a,b\}$ is called a <u>type 1</u> separation pair. The type 1 pairs for the graph in Figure 4 are (1,3),(1,4),(1,5),(4,5),(1,8).)

(2) There is a vertex $r \neq b$ such that $a \rightarrow r \overset{*}{\rightarrow} b$; b is a first descendant of r (i.e. a, r, and b lie on a common generated path); $a \neq 1$; every frond $x \dashrightarrow y$ with $r \leq x < b$ has $a \leq y$; and every frond $x \dashrightarrow y$ with $a < y < b$ and $b \rightarrow w \overset{*}{\rightarrow} x$ has LOWPT1$(w) \geq a$. ($\{a,b\}$ is called a <u>type 2</u> separation pair. The type 2 pairs for the graph in Figure 4 are (4,5) and (8,12)).

(3) (a,b) is a multiple edge of G and G contains at least four edges.

<u>Proof</u>: The converse part of the lemma is easiest to prove. Suppose pair $\{a,b\}$ satisfies (1), (2), or (3). Let E_i for $1 \leq i \leq k$ be the separation classes of G with respect to $\{a,b\}$. Suppose $\{a,b\}$ satisfies (1). Then the edge (b,r) is contained in some separation class, say E_1. Every tree arc with an endpoint in $D(r)$ has the other endpoint in $D(r) \cup \{a,b\}$. Also, since LOWPT1$(r)=a$ and LOWPT2$(r) \geq b$, every frond with an endpoint in $D(r)$ has the other endpoint in $D(r) \cup \{a,b\}$. Thus E_1 consists of all edges with an endpoint in $D(r)$. No other edges are in E_1 and the edges incident to vertex s must be in some other class, say E_2. Since E_1 and E_2 each contain two or more edges, $\{a,b\}$ is a separation pair.

Suppose $\{a,b\}$ satisfies (2). Let $S=D(r)-D(b)$. All edges incident to a vertex in S are in the same separation class, say E_1. Since b is a first descendant of r, $S=\{x \mid r \le x < b\}$ by Lemma 1. Let $b_1, b_2, \ldots, b_n$ be the sons of b in the order they occur in $A(b)$. Let $i_0 = \min\{i \mid \mathrm{LOWPT1}(b_i) \ge a\}$. By the ordering imposed on A, $i < i_0$ implies $\mathrm{LOWPT1}(b_i) < a$, and $i \ge i_0$ implies $\mathrm{LOWPT1}(b_i) \ge a$. By (2), every frond with tail in S has its head in $S \cup \{a\}$. Also by (2), every frond with head in S has its tail in $S \cup \{b\} \cup (\bigcup_{i \ge i_0} D(b_i))$. Every edge with an endpoint in $D(b_i)$, $i \ge i_0$, has its other endpoint in $S \cup \{a,b\} \cup D(b_i)$. Thus the class E_1 contains <u>at least</u> all edges with an endpoint in S, and <u>at most</u> all edges with an endpoint in $S \cup \bigcup_{i > i_0} D(b_i)$. Since $a \ne 1$, the edges incident to the root of P cannot be in E_1, and therefore $\{a,b\}$ is a separation pair.

Now we must prove the direct part of the lemma. Thus suppose that $\{a,b\}$ is a separation pair with $a<b$. If (a,b) is a multiple edge of G then it is clear that $\{a,b\}$ satisfies (3). Thus suppose that (a,b) is not a multiple edge of G. By Lemma 12, $a \overset{*}{\rightarrow} b$. Let E_i for $1 \le i \le k$ be the separation classes of G with respect to $\{a,b\}$. Let v be the son of a such that $a \rightarrow v \overset{*}{\rightarrow} b$, $S = D(v) - D(b)$, and $X = V - D(a)$. (Either S or X or both may be empty.) $E(S)$ and $E(X)$ are each contained in a separation class, say $E(S) \subseteq E_1$ and $E(X) \subseteq E_2$.

Let $a_i \neq v$ be a son of a. If a has such a son, LOWPT1(a_i)<a. This means that $E(D(a_i)) \subseteq E_2$. Let $Y = X \cup \bigcup_i D(a_i)$. Let $b_1, b_2, \ldots, b_n$ be the sons of b in the order they occur on the adjacency list of b. Let $E(D(b_i))$ be the set of edges with an endpoint in $D(b_i)$. The separation classes must be unions of the sets $E(S)$, $E(Y)$, $\{(a,b)\}$, $E(D(b_1))$, $E(D(b_2)), \ldots, E(D(b_n))$.

If $E(D(b_i)) = E_j$ for some i and j, then LOWPT1(b_i)=a since G is biconnected and this means LOWPT1(b_i)<b by Lemma 7. Also, LOWPT2(b_i)$\geq$b. Since $\{a,b\}$ is a separation pair, there must be a separation class other than E_j and $\{(a,b)\}$. Thus there is a vertex s such that $s \neq a$, $s \neq b$, and $s \notin D(b_i)$. This means that $\{a,b\}$ satisfies (1) where r is b_i.

Suppose now that no $E(D(b_i))$ is by itself a separation class. Let $i_0 = \min\{i \mid \text{LOWPT1}(b_i) \geq a\}$. If $i \geq i_0$ then since G is biconnected it must be the case that LOWPT1(b_i)<b, and the separation classes are $E_1 = E(S) \cup \bigcup_{i \geq i_0} E(D(b_i))$, $E_2 = E(Y) \cup \bigcup_{i < i_0} E(D(b_i))$, $E_3 = \{(a,b)\}$. (E_3 may be empty.) We have $v \neq b$ since $\{a,b\}$ is not a type 1 pair and $a \neq 1$ since E_2 is non-empty. If $x \dashrightarrow y$ is a frond with $v \leq x < b$, then $x \in S$, $(x,y) \in E_1$, and $a \leq y$. If $x \dashrightarrow y$ is a frond with $a < y < b$ and $b \rightarrow b_i \overset{*}{\rightarrow} x$, then $y \in S$, $(x,y) \in E_1$, and $i \geq i_0$, which means that LOWPT1(b_i)$\geq$a. We must verify one more condition to show that (2) holds; namely, that b is a first descendant of v. Since G is biconnected, LOWPT1(v) < a. Thus some frond with tail in D(v) has head less than a.

By the ordering imposed on A and the definition of a first descendant, there exists some frond $x \dashrightarrow y$ with $x \in D(v)$ and $y<a$ such that x is a <u>first</u> descendant of v. If b were not a first descendant of v then x would be in S, and E_1 and E_2 could not be distinct separation classes. Thus b is a first descendant of v, and (2) holds with r=v. This completes the proof of the direct part of the lemma.

Lemma 13 and its proof are worth pondering carefully. The lemma gives three easy-to-apply conditions for separation pairs. Conditions (1) and (2) identify the non-trivial separation pairs of the multigraph. Condition (3) handles multiple edges. Condition (1) requires that a simple test be performed on each tree arc of P. Thus testing for type 1 pairs requires O(V) time. Testing for type 2 pairs is somewhat harder but may be done in O(V+E) time using another depth-first search. Let {a,b} be a type 2 pair satisfying $a \to r \xrightarrow{*} b$ and $i_0=\min\{i \mid LOWPT2(b_i) \geq a\}$, where $b_1, b_2, \ldots, b_n$ are the sons of b in the order they occur in A(b), then one separation class with respect to {a,b} is $E(\{x \mid r \leq x < b_{i_0} + ND(b_{i_0})\} - \{b\})$. This follows from the proof of Lemma 13. The new numbering, which satisfies the somewhat strange condition in Lemma 9, thus makes it easy to determine the separation classes and to divide the graph once a separation pair is found. An algorithm for finding split components based on Lemma 13 is given in the next section.

Finding Split Components

We find split components by examining the generated paths in order and testing for separation pairs with Lemma 13. Separation pairs will be of several types. Multiple edges and type 1 pairs are easy to recognize. So are type 2 pairs $\{a,b\}$ where $a \to v \to b$ and v has degree two. Other type 2 pairs are somewhat harder to recognize. Let c be the first path generated (a cycle). The cycle consists of a set of tree arcs $1 \to v_1 \to v_2 \to \cdots \to v_n$ followed by a frond $v_n \dashrightarrow 1$. The vertex numbering is such that $1 < v_1 < \ldots < v_n$. When c is removed, the graph falls into several connected pieces, called segments. Each segment consists either of a single frond (v_i, v_j) or of a tree arc (v_i, w) plus a subtree with root w plus all fronds leading from the subtree. The order of path generation is such that all paths in one segment are generated before paths in any other segment, and the segments are explored in decreasing order of v_i.

Suppose we repeat the pathfinding search, using it now to find split components. We shall keep a stack of edges, adding edges to this stack as we back up over them during the search. Each time we find a separation pair we remove a set of edges from the stack corresponding to a split component. We add a virtual edge corresponding to the split to both the component and to the edge stack. We also need to update various pieces of information, since the fathers of vertices and the degrees of vertices may change when a graph is split. The complete pathfinding search will create a complete set of split components. Assembling the split components to give the triconnected components is then a simple matter.

To identify type 2 pairs, we keep a stack (called TSTACK) of triples (h,a,b). {a,b} is a possible type 2 pair and h denotes the largest numbered vertex in the corresponding split component. The pairs are in nested order on the stack; that is, if v_i is the current vertex being examined by the pathfinding search and (h_1,a_1,b_1), $(h_2,a_2,b_2),\cdots,(h_k,a_k,b_k)$ are on TSTACK then $a_k \leq a_{k-1} \leq \cdots \leq a_2 \leq a_1 \leq v_i \leq b_1 \leq b_2 \leq \cdots \leq b_k$. Furthermore all the a_j and b_j are vertices on the cycle c.

We update TSTACK in the following way:

(1) each time we traverse a new path $p: s \overset{*}{=}> f$, we delete all triples (h_j,a_j,b_j) on top of the stack with $a_j > f$. If p has second vertex $v \neq f$, let $x = v + ND(v) - 1$. Otherwise let $x = s$. Let $y = \max\{h_j \mid \text{triple}(h_j,a_j,b_j) \text{ deleted from TSTACK}\}$. If (h_k,a_k,b_k) was the last triple deleted, we add $(\max(x,y),f,s)$ to the stack. If no triple was deleted, we add (x,f,s) to the stack.

(2) When we back up over a tree arc $v_i \rightarrow v_{i+1}$ with $v_i \neq 1$, we delete all entries (h_j,a_j,b_j) on top of TSTACK satisfying $\mathrm{HIGHPT}(v_i) > h_j$. This test is necessary to guarantee that entries not corresponding to type 2 pairs don't accumulate on TSTACK.

We use TSTACK to find separation pairs in the following way: whenever we back up along a tree arc $v_i \rightarrow v_{i+1}$ during the pathfinding search, we examine the top triple (h_1,a_1,b_1) on TSTACK. If $v_i \neq 1$, $a_1 = v_i$, and $a_1 \neq \mathrm{FATHER}(b_1)$, $\{a_1,b_1\}$ is a type 2 separation pair. If DEGREE(v_{i+1})=2 and v_{i+1} has a son, v_i and the son of v_{i+1} form a type 2 separation pair. We split off components corresponding to type 2 pairs until these two conditions give us no more components. (Simultaneously, we test for components corresponding to multiple edges and split these off.) Then we apply Lemma 13 to test whether $\{v_i$, LOWPT1(v_{i+1})$\}$ is a type 1 pair, splitting off a component if necessary. (Again, we need to check for a multiple-edge component.)

We handle the recursive part of the algorithm in the following way: Traversing a path $p:s \overset{*}{\Rightarrow} f$ which starts on c means the search is entering a new segment. Vertex f must be the lowest vertex in the segment by the ordering imposed on the pathfinding search. After we update TSTACK as described above, if p contains more than one edge we place an end-of-stack marker on TSTACK and continue finding paths. This corresponds to a recursive call of the basic triconnectivity algorithm. When we back up over the first edge of p, we delete entries from TSTACK all the way down to the end-of-stack marker. This corresponds to popping up from the recursion.

One point remains mysterious; namely, the reason we use LOWPT2 as well as LOWPT1 to construct A, the acceptable adjacency structure which determines the pathfinding search order. This step is necessary so that all multiple edges are handled correctly. Suppose v is a vertex, and w_1, w_2,...,w_k are the sons of v such that LOWPT1(w_i)=u. Further suppose that $v \dashrightarrow u$.

Let the w_i be ordered as in A(v). There is some i_0 such that $i \le i_0 \Rightarrow \mathrm{LOWPT2}(w_i) < v$ and $i > i_0 \Rightarrow \mathrm{LOWPT2}(w_i) \ge v$. In A(v), u will appear after all the w_i with $1 \le i \le i_0$. If $i > i_0$, $\{u, w_i\}$ is a type 1 separation pair; splitting off the corresponding component produces a new (virtual) frond $v \dashrightarrow u$. It is important that all the w_i with $i > i_0$ appear together in A(v) so that these virtual fronds may be located and combined to give split components which are bonds.

Below is an Algol-like procedure to find split components based on the ideas outlined above. The procedure is applicable to any biconnected multigraph for which steps 1, 2, and 3 described in the previous section have been carried out.

```
procedure SPLIT(G); begin
  comment procedure to determine split components of G, a bi-
    connected multigraph on which steps 1, 2 and 3 have been
    carried out.  G is represented by a set of properly or-
    dered adjacency lists A(v).  TSTACK contains triples re-
    presenting possible type 2 separation pairs.  ESTACK con-
    tains edges backed up over during search.  Other variables
    have been defined in the previous section;
  procedure PATHSEARCH(v); begin
    comment this recursive procedure repeats the pathfinding
      search, finding separation pairs and splitting off com-
      ponents as it proceeds.  It is based on the material in
      this section and the last.  Vertex v is the current ver-
      tex in the depth-first search;
    for w∈A(v) do
     if v → w then begin
A:     if v → w is first edge of a path then begin
         y:=0;
         while (h,a,b) on TSTACK has a>LOWPT1(w) do begin
           y:=max(y,h);
           delete (h,a,b) from TSTACK;
         end;
         if no triples deleted from TSTACK then add (w+ND(w)-1,
           LOWPT1(w),v) to TSTACK
         else if (h,a,b) last triple deleted then
           add (max{y,w+ND(w)-1}, LOWPT1(w),b) to TSTACK;
         add end-of-stack marker to TSTACK;
       end;
       PATHSEARCH(w);
       add (v,w) to ESTACK;
B:     while v≠1 and ((DEGREE(w)=2) and (A1(w)>w) or (h,a,b) on
           TSTACK satisfies (v=a)) do begin comment test for
           type 2 pairs;
```

```
    if (h,a,b) on TSTACK has (a=v) and (FATHER(b)=a)
      then delete (h,a,b) from TSTACK;
    else begin
      if (DEGREE(w)=2) and (A1(w)>w) do begin
        j=j+1;
        add top two edges (v,w) and (w,x) on ESTACK to
          new component;
        add (v,x,j) to new component;
        if (y,z) on ESTACK has (y,z)=(x,v) then begin
          FLAG:=true;
          delete (y,z) from ESTACK and save;
        end;
E:  end else if (h,a,b) on TSTACK satisfies v=a and
        a≠FATHER(b) then begin
      j=j+1;
      delete (h,a,b) from TSTACK;
      while (x,y) on ESTACK has (a≤x≤h) and (a≤y≤h) do
        if (x,y)=(a,b) then begin
          FLAG:=TRUE;
          delete (a,b) from TSTACK and save;
      end else begin
        delete (x,y) from ESTACK and add to current
          component;
        decrement DEGREE(x), DEGREE(y);
      end
      add (a,b,j) to new component;
      x:=b;
    end;
    if FLAG then begin
      FLAG:=false;
      j:=j+1;
      add saved edge,(x,v,j-1), (x,v,j) to new component;
      decrement DEGREE(x), DEGREE(v);
    end;
```

```
      add (v,x,j) to ESTACK;
      increment DEGREE(x), DEGREE(v);
      FATHER(x):=v;
      if A1(v)≠x then A1(v)=x;
      w:=x;
    end;
    comment test for a type 1 pair;
G:  if (LOWPT2(w)≥v) and ((LOWPT1(w)≠1) or (FATHER(v)≠1)
        or (w>3))
      then begin
        j:=j+1;
        while(x,y) on top of ESTACK has (w≤x<w+ND(w)) or
            (w≤y<w+ND(w))
          then begin
            delete (x,y) from ESTACK;
            add (x,y) to new component;
            decrement DEGREE(x), DEGREE(y);
          end;
        add (v, LOWPT1(w), j) to new component;
        if A1(v)=w then A1(v):=LOWPT1(w);
        comment test for multiple edge;
        if (x,y) on top of ESTACK has (x,y)=(v,LOWPT1(w))
            then begin
          j:=j+1;
          add (x,y), (v,LOWPT1(w),j=1), (v,LOWPT1(w),j) to
            new component;
          decrement DEGREE(v), DEGREE(LOWPT1(w));
        end;
        if LOWPT1(w)≠FATHER(v) then begin add (v,LOWPT1(w),j)
            to ESTACK;
          increment DEGREE(v), DEGREE(LOWPT1(w));
        end else begin
          j:=j+1;
          add (v,LOWPT1(w), j-1), (v,LOWPT1(w),j), tree
            arc (LOWPT1(w),v) to new component;
```

```
            mark tree arc (LOWPT1(w),v) as virtual edge j;
          end;
        end;
C:      if v→w is first edge of a path then delete all entries
          on TSTACK down to and including end-of-stack marker;
D:      while (h,a,b) on ESTACK has HIGHPT(v)>h, do delete(h,a,b)
          from TSTACK;
      end else begin comment v-→w;
F:      if v-→w is first (and last) edge of a path then begin
          y:=0;
          while (h,a,b) on TSTACK has a>w do begin
            y:=max(y,h);
            delete (h,a,b) from TSTACK;
          end;
          if no triples deleted from TSTACK then add (v,w,v) to
            TSTACK;
          if (h,a,b) last triple deleted then add (y,w,b) to
            TSTACK;
        end;
        if w=FATHER(v) then begin
          j:=j+1;
          add (v,w), (v,w,j), tree arc (w,v) to new component;
          decrement DEGREE(v), DEGREE(w);
          mark tree arc (w,v) as virtual edge j;
        end else add (v,w) to ESTACK;
  end; end;
  j:=0;
  FLAG:=false;
  PATHSEARCH(1);
end;
```

Lemma 14: SPLIT correctly divides a biconnected multigraph G into split components.

Proof: We must prove two things: (1) if G is triconnected, SPLIT will not split it, and (2) if G is not triconnected, the algorithm will split it. Once we have these two facts, we may prove the lemma by induction on the number of edges in the graph. The tests for multiple edges, for type 1 separation pairs, and for degree two vertices are straightforward. (The type 1 test (G in PATHSEARCH) includes the condition (LOWPT1(w)$\neq$1) or (FATHER(v)$\neq$1) or (w>3) to make sure that some vertex lies outside the corresponding split component.) These tests will discover a separation pair of the correct type if one exists, and they will not report a separation pair if one does not exist. Thus we must only show that the type 2 test works correctly on multigraphs with no degree two vertices, multiple edges or type 1 separation pairs, and we will have verified (1) and (2).

Suppose G is a biconnected multigraph with no degree 2 vertices, multiple edges, or type 2 separation pairs. Let us consider the type 2 test and the changing contents of TSTACK as the search of G progresses. If $(h_1,a_1,b),\ldots,(h_k,a_k,b_k)$ are the contents of TSTACK above the highest end-of-stack marker and if v is the vertex currently being examined during the search, then $a_k \leq a_{k-1} \leq \cdots \leq a_1 \leq v \leq b_1 \leq \cdots \leq b_k$. This follows by induction from an examination of the possible changes that can be made in TSTACK(statements A,B,C,D,E,F in PATHSEARCH). Furthermore $a_k, a_{k-1} \cdots v$, $b_1 \cdots b_k$ all lie on the cycle corresponding to the current recursive call of the basic triconnectivity algorithm.

Suppose (h,a,b) on TSTACK is found to satisfy the type 2 test when the search returns along a tree arc $v \to w$. The test (B, E in PATHSEARCH) states that $a=v$, $v \neq 1$, and FATHER(b)$\neq a$. It follows that $r=A1(a) \neq b$ satisfies $a \to r \overset{*}{\to} b$ and b is a first descendant of r (that is, a,r, and b lie on a common generated path). If some frond $x \dashrightarrow y$ with $r \leq x < b$ had $a > y$, the triple on TSTACK corresponding to (h,a,b) would have been deleted from TSTACK when the frond was explored (A or F in PATHSEARCH). Similarly, if some frond $x \dashrightarrow y$ with $a < y < b$ and $b \to w \overset{*}{\to} x$ had LOWPT1(w) $<a$, the triple on TSTACK corresponding to (h,a,b) would have been deleted by the HIGHPT test when vertex y was examined (D in PATHSEARCH). It follows that {a,b} is a type 2 separation pair by Lemma 13.

Conversely, suppose G has a type 2 pair {a,b}. Let $b_1, \cdots, b_n$ be the sons of b in the order they occur in A(b). Let $i_0 = \min\{i \mid \text{LOWPT1}(b_i) \geq a\}$. If i_0 exists, then (b_{i_0}+ND(b_{i_0}), LOWPT1(b_i),b) will be placed on TSTACK when tree arc $b \to b_i$ is explored. This triple may be deleted from TSTACK, but it will always be replaced by a triple of the form (h,x,b) with LOWPT1(b_i)$\geq x \geq a$. Eventually such a triple will satisfy the type 2 test, unless some other type 2 pair is found first. If i_0 does not exist, let (i,j) be the first edge traversed after b is reached such that $a \le i$ and $j \le b$. If $i \dashrightarrow j$ then (i,j,i) will be placed on TSTACK, possibly modified, and eventually selected as a type 2 pair, unless some other type 2 pair is found first. If $i \to j$ then (j+ND(j), LOWPT1(j),i) will be placed on TSTACK, possibly modified, and eventually selected as a type 2 pair unless some other type 2 pair is found first. Thus if some type 2 pair exists, at least one type 2 pair will be found by the algorithm. it follows that the type 2 test works correctly, and the algorithm splits a multigraph if and only if a separation pair exists.

The lemma follows by induction on the number of edges in G. Suppose the lemma is true for graphs with fewer than k edges. Let G have k edges. If G cannot be split, the algorithm works correctly on G by the argument above. If G can be split, it will be split. Consider the first split performed by the algorithm, producing split graphs G_1 and G_2. The behavior of the algorithm on G is a composite of its behavior on G_1 and G_2. Since the algorithm splits G_1 and G_2 correctly by the induction hypothesis, it must split G correctly. The lemma follows by induction. Figure 4 gives the contents of ESTACK and TSTACK when the first separation pair (8,12) in the graph of Figure 1 is detected.

Lemma 15: The triconnected components algorithm processes a graph G with V vertices and E edges in O(V+E) time.

Proof: The number of edges in a set of split components of G is bounded by 3E-6 by Lemma 1. All steps except finding split components thus require O(V+E) time by the results of the last two sections. Consider execution of algorithm SPLIT. Each edge is placed on ESTACK once and deleted once. The depth-first search itself requires O(V+E) time, including the various tests. The number of triples added to TSTACK is O(V+E). Each triple may only be modified if it is on top of the stack. Thus the time necessary to maintain TSTACK is also O(V+E) and SPLIT requires O(V+E) time.

This completes our presentation of an O(V+E) triconnected components algorithm. This algorithm may be used in the construction of an O(V log V) algorithm for testing isomorphism of planar graphs [3]. The algorithm is not only theoretically optimal (to within a constant factor) but practically useful. The split components algorithm has been implemented in ALGOL W and run on an IBM 360/65 computer. Experiments show that the algorithm can handle graphs with around 1000 edges in less than 10 seconds.

References

[1] A. Ariyoshi, I. Shirikawa, and O. Hiroshi, "Decomposition of a Graph into Compactly Connected Two-Terminal Subgraphs", IEEE Trans. on Circuit Theory, Vol.CT-18 (1971) 430-435.

[2] J. Bruno, K. Steiglitz, and L. Weinberg, "A New Planarity Test Based on 3-Connectivity", IEEE Trans. on Circuit Theory, Vol. CT-17 (1970), 197-206.

[3] J. Hopcroft and R. Tarjan,"Isomorphism of Planar Graphs", Complexity of Computer Computations, Plenum Press, New York, 1972, 143-150.

[4] Ledergerg, J. "DENDRAL-64:A System for Computer Construction, Enumeration, and Notation of Organic Molecules as Tree Structures and Cyclic Graphs, Part II: Topology of Cyclic Graphs," Interim Report ot the National Aeronautics and Space Administration, Grant Nos. 681-60, NASA CR 68898, STAR No. N-66-14074 (December, 1965).

[5] R. Tarjan, "Depth-First Search and Linear Graph Algorithms", SIAM J. COMPUT., Vol.1, No.2 (June, 1972), 146-159.

[6] J. Hopcroft and R. Tarjan, "Efficient Algorithms for Graph Manipulation", Comm. ACM, to appear.

[7] R. Tarjan, "An Efficient Planarity Algorithm", STAN-CS-244-71, Computer Science Dept., Stanford University (November, 1971).

[8] J. Hopcroft and R. Tarjan, "Efficient Planarity Testing", Dept. of Computer Science, Cornell University,to appear.

[9] R. Tarjan, "Finding Dominators in Directed Graphs", Dept. of Computer Science, Cornell University, to appear.

[10] D.J. Kleitman,"Methods for Investigating the Connectivity of Large Graphs", IEEE Trans. on Circuit Theory, Vol. CT-16(1969), 232-233.

[11] S.A. Cook, "Linear-Time Simulation of Deterministic Two-Way Pushdown Automata", IFIP Congress 71: Foundations of Information Processing, Ljubljana, Yugoslavia (August, 1971), North Holland Pub. Co., Amsterdam 174-179.

[12] G. Busacker and T.L. Saaty, Finite Graphs and Networks: An Introduction with Applications, McGraw-Hill, New York, 1965.

[13] F. Harary, Graph Theory, Addison-Wesley, Reading, Mass., 1969.

[14] S. Maclaine, "A Structural Characterization of Planar Combinatorial Graphs", Duke Math. Journal, Vol. 3, No. 3,(September, 1937), 460-472.

[15] W.T. Tutte, Connectivity in Graphs, University of Toronto Press, 1966.

[16] J. Edmonds and W. Cunningham, private communication.

[17] R. Tarjan and J. Hopcroft, "Finding the Triconnected Components of a Graph", TR 72-140, Department of Computer Science, Cornell University, August 1972.

[18] L. Auslander and S.V. Parter, "On Imbedding Graphs in in the Plane", Journal of Mathematics and Mechanics, Vol. 10, No.3 (May, 1961), 517-523.

[19] A.J. Goldstein, "An Efficient and Constructive Algorithm for Testing Whether a Graph Can Be Embedded in a Plane", Graph and Combinatories Conference, Office of Naval Research Logistics Proj., Contract No. NONR 1858-(21) Dept. of Math., Princeton University (May 16-18, 1963) 2 unno.pp.