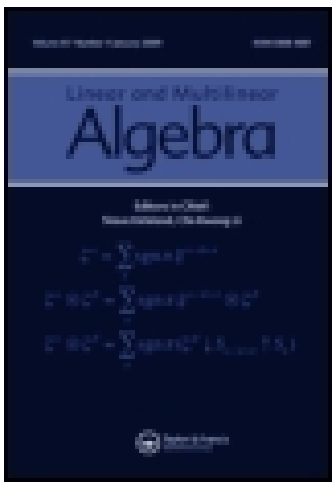


Linear and Multilinear Algebra
Publication details, including instructions for authors and subscription information:
http://www.tandfonline.com/loi/glma20

Finding triconnected components of graphs

Kiem-Phong Vo [a]
[a] Bell Laboratories , 600 Mountain Ave., Murray Hill, NJ, 07974
Published online: 30 May 2007.

Linear and Multilinear Algebra, 1983, Vol. 13, pp. 143–165
0308–1087/83/1302–0143 $18.50/0

Printed in the United States of America

Finding Triconnected Components of Graphs

KIEM-PHONG VO
Bell Laboratories, 600 Mountain Ave., Murray Hill, NJ 07974

(*Received October 11, 1981*)

I. INTRODUCTION

Determining connectivity of graphs is an interesting and important problem in graph theory. Although polynomial time algorithms for determining the connectivity of a graph exist [3, 4], the optimal complexity of an algorithm for such a purpose is still unknown. Since any algorithm that determines connectivity has to examine all edges of a graph, it is clear that the algorithm will require at least linear time in the number of edges. In this sense, there are optimal linear time algorithms to determine low connectivities. Most of the early works in this area are done by Hopcroft and Tarjan [1, 6, 8]. Hopcroft and Tarjan successfully used the depth-first search technique to construct linear time algorithms for testing planarity, finding 1, 2, 3-connected components and many other interesting properties of graphs.

In [10], we gave a characterization of triconnectivity based on the structure of segment graphs of cycles in a depth-first cycle basis. This characterization leads to a new linear time algorithm to determine triconnected components of a graph which will be described subsequently. This algorithm is similar in spirit to the Hopcroft–Tarjan algorithm [6]. Our new contribution is the way the computation is organized in terms of segment graphs so that the process can be visualized more clearly and graphically.

II. BASIC OPERATIONS AND OUTLINES OF THE ALGORITHM

First we remind the readers a few important definitions given in [10]. Let $G = (V, E)$ be multigraph (i.e., G may have multiple edges). Let $H = (W, F)$ be a subgraph of G, we define an equivalence relation on $E - F$ as follows:

DEFINITION 1 a. For all $e \in E - F$, $e \cong e$, b. For all $e, f \in E - F$, $e \cong f$ iff there exists a simple path joining e, f that has no internal vertex contained in W.

It is not too hard to see that $\cong$ is an equivalence relation. If H is a cycle, the equivalence classes are called *segments* of H. On the other hand, if $H = (\{a, b\}, \phi)$, the equivalence classes are called *separation classes* relative to the pair $\{a, b\}$ [9].

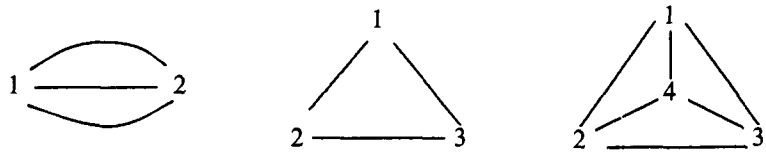
DEFINITION 2 Let $S_1, S_2, \ldots, S_k$ be the separation classes of a pair $\{a, b\}$. If there is a partition (A, B) of $\{1, 2, \ldots, k\}$ so that $|E_1 = \bigcup_{i \in A} S_i| \geqslant 2$ and $|E_2 = \bigcup_{j \in B} S_j| \geqslant 2$, we say that $\{a, b\}$ is a *separation pair* (see also II.7 in [10]).

We note that if $H_1 = (V(E_1), E_1)$ and $H_2 = (V(E_2), E_2)$ then $V(H_1) \cap V(H_2) = \{a, b\}$.

DEFINITION 3 If G is biconnected and G has no separation pair then G is *triconnected* (see II.8 in [10]).

It is easy to see that the only triconnected graph with multiple edges is the *3-bond* where a *k-bond* is a graph with 2 vertices and k edges joining them. It is equally clear that the only triconnected graph with degree two vertices is the triangle or the complete graph on 3 vertices.

Example 1 Three small triconnected graphs:



From the definition, each triconnected subgraph of G must be contained in a biconnected component of G. The connected and biconnected components can be easily detected in linear time [1, 8]. Thus, from now on to the end of this paper we shall fix G as a biconnected graph.

1. THE SPLITTING OPERATION

Suppose that G has a separation pair $\{a,b\}$. Let H_1, H_2 be subgraphs of G as in defintion 2. Then $|E(H_1)| \geqslant 2$, $|E(H_2)| \geqslant 2$ and $V(H_1) \cap V(H_2) = \{a,b\}$. Let $G_i = H_i + (a,b)$ for $i = 1,2$. The G_i's are called *split graphs* of G at $\{a,b\}$. One should note immediately that the split graphs are not unique because there are usually many ways to partition the separation classes of $\{a,b\}$. The new edge (a,b) is called a *virtual edge*. We shall think of a virtual edge as a triple (a,b,n) with n being some label to distinguish this edge from other edges. It can be seen easily that the split graphs are again biconnected. Thus, we can split the split graphs then their split graphs and so on until no more splitting is possible. The obtained graphs divide into three classes, the triple bonds, the triangles and triconnected graphs with more than 3 edges. These are called the *split components* of G. *They are not unique.*

The following result is due to Hopcroft and Tarjan [6].

LEMMA 1 *The total number of edges in all the split components does not exceed* $3|E| - 6$.

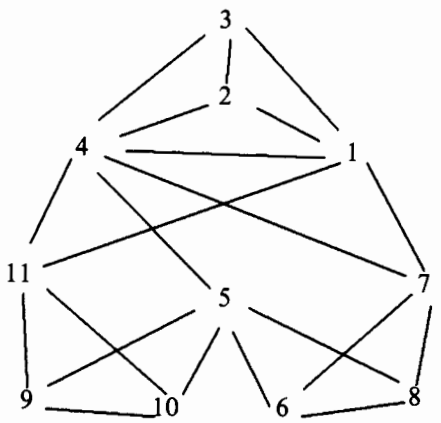
2. THE MERGING OPERATION

Let H_1 and H_2 be two split graphs of G that contain the same virtual edge (a,b,n). We merge them by letting $H = H_1 + H_2 - (a,b,n)$. Merging is the inverse of splitting.

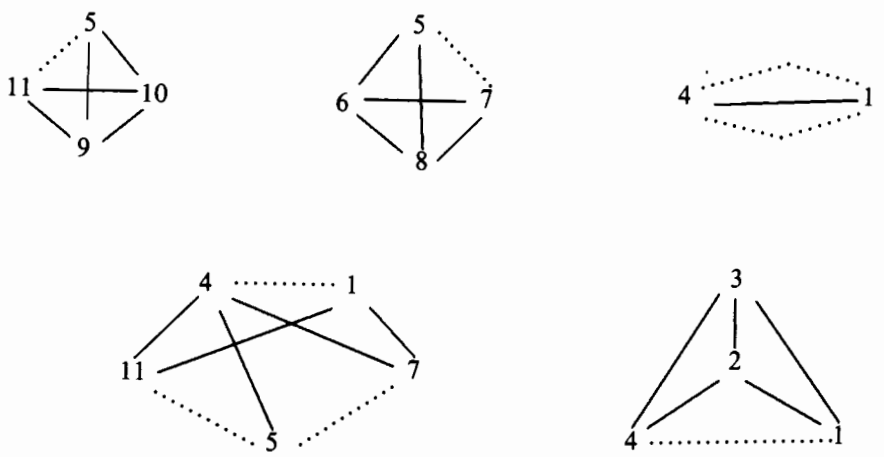
Now suppose that we have split G into split components. Then the 3-bonds are merged as much as possible to form a set of bonds and the triangles are merged as much as possible to form a set of polygons. The final set of graphs is called a *complete set of triconnected components* or *tricomponents* or G. We have [6, 7]:

LEMMA 2 *A complete set of tricomponents is unique up to isomorphism.*

Example 2 The following biconnected graph G has three separation pairs: $\{1,4\}$, $\{5,7\}$ and $\{11,5\}$.



Following is a set of split components of G:



3. OUTLINES OF THE TRICOMPONENT DETECTION ALGORITHM

The tricomponent algorithm is organized in three main steps as follows:

a. Split off multiple edges to form 3-bonds and a simple biconnected graph G'.

b. Find split components of G'.

c. Merge 3-bonds and triangles into bonds and polygons.

Steps (a) and (c) are easy and can be done in $O(|V| + |E|)$ time. For step (a), we simply order all edges of G lexicographically using a bucket sort [1]. Multiple edges will end up next to each other and can be split off easily. Step (c) is somewhat harder. During splitting, we build auxiliary graphs with vertices being the virtual edges and the bonds (or triangles). Edges join bonds (triangles) to the virtual edges that they contain. At the end, we simply detect the connected components of the resulting graphs and merge the appropriate vertices. Note that lemma 1 guarantees that this can be done in $O(|E|)$ time. Step (b) remains the most difficult and interesting part of the algorithm, the rest of the paper is devoted to solving it.

III. DEPTH-FIRST SEARCH AND CONSTRUCTION OF PATH TREES.

From here to the end of this paper we shall assume that $G = (V, E)$ is a biconnected simple graph. There are many different ways to represent a graph. For efficient implementation of depth-first search, we assume that graphs are represented by their adjacency lists [1, 6, 11]. So each vertex v in the graph is associated with a linearly ordered list $A(v)$ of all vertices adjacent to it.

1. LINEAL NUMBERING AND LOW POINTS

A connected graph T without any cycle is called a *tree*. If in addition a vertex r is singled out in T, T is called *rooted tree* with root r. So a rooted tree is a pair (T, r) with T a tree and r the root of T. In a rooted tree (T, r), we can assign direction to edges by always directing away from the root [10, 11]. Thus a rooted tree can always be considered a directed tree. Let v, w be two vertices in a rooted tree T. We say that v is an *ancestor* of w if there exists a directed path from v to w in T. We also say that w is a *descendant* of v and v, w are *lineally related*. Lineal relation defines a partial order on the set of vertices of the tree. The order is $v \leqslant w$ iff v is an ancestor of w. It is clear that the root is the minimal element in this order.

Let (T,r) be any rooted tree with vertex set W. A *lineal numbering* of (T,r) is a bijection $\mathrm{LN}: W \leftrightarrow \{1,2,\ldots,|W|\}$ that is compatible with the lineal order of T. That is, $\mathrm{LN}(v) \leqslant \mathrm{LN}(w)$ whenever $v \leqslant w$ in the lineal order.

A subgraph $T = (V, E_T)$ of G is a *spanning tree* if T is a tree. A rooted spanning tree $T = (V, E_T, r)$ of G is said to be *lineal* if for all edges $(s,t) \in E - E_T$, s and t are lineally related. Such an edge (s,t) is directed from s to t if $t \leqslant s$ in the lineal order. Then, (s,t) is called a *backedge*. Let $E_B = E - E_T$. The subgraph of G with edge set E_B is called the *backedge graph* B.

Let (T,r) be a lineal spanning tree of G and B be the backedge graph. Suppose that we have a lineal numbering LN of the vertices of G, i.e., a numbering consistent with the lineal order induced by (T,r). For any edge $e = (a,b)$ in E_T, let $R(e)$ be the set consisting of all $\mathrm{LN}(t)$ where (s,t) is a backedge and s is a descendant of b including b. Now for all edges $e \in E$, we define LOW1(e), LOW2(e) (the first and second low points of e [6, 8, 11]) as:

DEFINITION 4 a. If $e = (a,b) \in E_T$, then:

$$\mathrm{LOW1}(e) = \min\{\mathrm{LN}(b)\} \cup R(e),$$

$$\mathrm{LOW2}(e) = \min\{\mathrm{LN}(b)\} \cup ((R(e) - \{\mathrm{LOW1}(e)\}).$$

b. If $e = (s,t) \in E_B$, then:

$$\mathrm{LOW1}(e) = t,$$

$$\mathrm{LOW2}(e) = s,$$

We now give the depth-first search procedure that will decompose the graph G into a lineal spanning tree (T,r) and a backedge graph B. At the same time, the procedure also computes a lineal numbering, the low point values of edges and the FATHER function that gives the father of each vertex in the lineal spanning tree. Before the procedure is called, LOW1(e), LOW2(e) and LN(v) are initialized to zero for every edge e and vertex v. An external integer n is kept for the number of vertices that have been visited at any stage in the procedure. The procedure is invoked by calling DEPTH-FIRST(r,s) where (r,s) will be the first tree edge and r will be the root of the tree constructed. LN(r) and n are initialized to 1.

```
1)  procedure DEPTH-FIRST(u, v)
2)  begin
3)    n := n + 1;
4)    LOW1(u, v) := LOW2(u, v) := LN(v) := n;
5)    FATHER(v) := u;
6)    for each w ∈ A(v) do
7)      if LN(w) = 0 then
8)        begin
9)          add(v, w) to E_T;
10)         DEPTH-FIRST(v, w);
11)         ADJUST-LOWPT((u, v), (v, w));
12)       end
13)     else if w ≠ u then
14)       begin
15)         add(v, w) to E_B;
16)         LOW1(v, w) := LN(w);
17)         LOW2(v, w) := LN(v);
18)         ADJUST-LOWPT((u, v), (v, w));
19)       end
20) end
21) procedure ADJUST-LOWPT((u, v), (v, w))
22) begin
23)   if LOW1(v, w) < LOW1(u, v) then
24)     begin
25)       LOW2(u, v) := min{LOW2(v, w), LOW1(u, v)};
26)       LOW1(u, v) := LOW1(v, w);
27)     end
28)   else if LOW1(v, w) = LOW1(u, v) then
29)         LOW2(u, v) := min{LOW2(v, w), LOW2(u, v)};
30)     else LOW2(u, v) := min{LOW1(v, w), LOW2(u, v)};
31) end
```

Remarks

1: (u, v) is the tree edge being examined.

3: increase the vertex counter (v is the current vertex).

4–5: define LN(v), FATHER(v) and initialize the low point values of (u, v).

6: examine all edges incident on v.

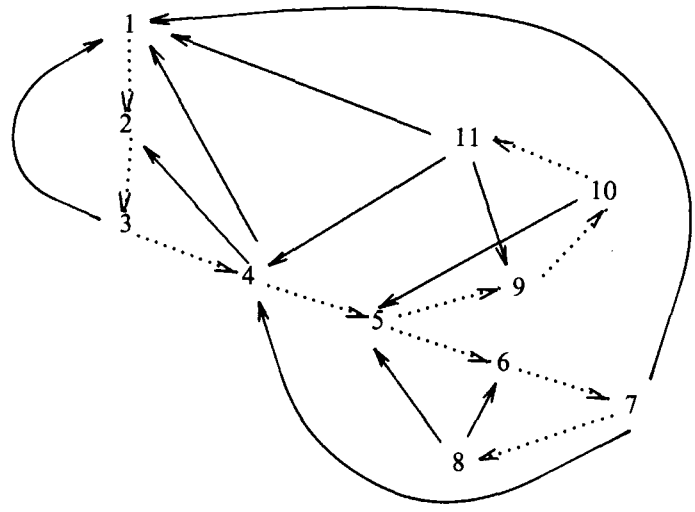
7–12: (v, w) is a new tree edge, make the basic recursive call of DEPTH-FIRST.

13–19: (v, w) is a new backedge.

21–32: recalculate low point values of (u, v) based on the values of the descendant edge (v, w).

Since each edge is traversed only once by DEPTH-FIRST, the time complexity of DEPTH-FIRST is linear in the number of edges. As LN is a bijection, we can and shall identify each vertex of G with its lineal number.

Example 3 Following is a decomposition of the graph G in example 2 by DEPTH-FIRST. The vertex numbering is also a lineal numbering.



Following is the list of directed edges of G and their low points:

EDGE	LOW1	LOW2	EDGE	LOW1	LOW2
(1,2)	1	1	(7,1)	1	7
(2,3)	1	2	(7,4)	4	7
(3,1)	1	3	(7,8)	5	6
(3,4)	1	2	(8,5)	5	8
(4,5)	1	4	(8,6)	6	8
(4,1)	1	4	(9,10)	1	4
(4,2)	2	4	(10,11)	1	4
(5,6)	1	4	(10,5)	5	10
(5,9)	1	4	(11,1)	1	11
(6,7)	1	4	(11,4)	4	11
			(11,9)	9	11

2. CONSTRUCTION OF A PATH TREE

Henceforth, we shall assume that G has been decomposed into a lineal spanning tree (T,r) and a directed backedge graph B, all vertices of G are identified by their lineal numbers, and their low points are known.

We associate with each directed edge $e=(a,b)$ a triple $(a, \text{LOW1}(e), x, (\text{LOW2}(e) < a))$ where:

$$x(\text{statement}) = \begin{cases} 1 & \text{if statement is true} \\ 0 & \text{if statement is false} \end{cases}$$

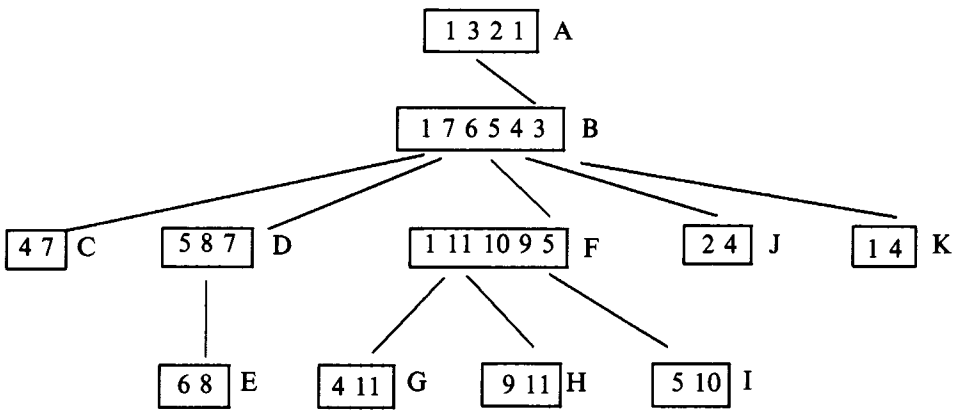
Using lexicographic bucket sort [1], we can order these triples in linear time. In this linear order of edges, T and B become ordered directed graphs. We denote these ordered directed graphs As and B^* respectively. $T^* + B^*$ is called a *proper decomposition* of G [11].

Given $e \in E$, we construct a path PATH(e) in G containing e as follows. Let e be the first edge of PATH(e). If e is not a backedge, let e' be the first unused edge of $T^* + B^*$ incident to e. Add e' to PATH(e) then repeat the process unless e' is a backedge. From the ordering of edges in $T^* + B^*$ it is not hard to see that if PATH(e) $= (e_1, e_2, \ldots, e_k)$ with $e_i = (a_i, b_i)$ then $b_k = \text{LOW1}(e)$.

Now we define recursively the ordered rooted tree of paths PATR(G,T,r) whose vertices form a subset of $\{\text{PATH}(e) : e \in E\}$. Since G is biconnected and (T,r) is lineal, there is a unique tree edge d incident to r in T [1]. Let PATH(d) be the root of PATR(G,T,r). In general if PATH(e) is a vertex of the path tree, let PATH(e) $= (e_1, e_2, \ldots, e_k)$ with $e_i = (a_i, b_i)$. For $i = 1, 2, \ldots, k$, let $E(e_i)$ be the ordered list (in $T^* + B^*$) of edges incident to a_i except e_i. Let SON(e) $= E(e_k)*(e_{k-1}) \ldots *E(e_2)$ where $*$ means concatenation. The ordered list of edges directed out from PATH(e) in PATR(G,T,r) is $\{(\text{PATH}(e), \text{PATH}(f)) : f \in \text{SON}(e)\}$. One should note that the order of concatenation is backward in the list of edges of PATH(e) and SON(e) is empty if e is a backedge. The vertices of PATR(G,T,r) form a partition of E, and PATH(d) is a simple cycle [6, 8, 11]. Further there is exactly one vertex of PATR(G,T,r) for each backedge of G. Using the function FATHER, one can easily complete each path PATH(e) into cycle, called CYCLE(e). The set of cycles resulted this way is a depth-first cycle basis for G. Thus, we see that the path tree is an instance of a cycle tree as considered in [10].

Example 4 The reader can easily verify that the list of edges in example 3 is sorted as discussed above. The path tree of G using this

sorted list is given below. The alphabets indicate the order in which the paths were generated.



IV. SEGMENT GRAPHS AND THE MAIN THEOREM

In this section, a path tree PATR(G, T, r) for G will be fixed. Most of the terminology and results covered here are given in [10]. This section rephrases them to provide an interface to the language pertinent to the split algorithm and to the paper of Williamson [11].

DEFINITION 5 Let PATH(e) be a nonroot node in PATR(G, T, r). SEG(e) is the union of PATH(e) and all of its descendants in the path tree.

As noted before, using the function FATHER, we can easily complete every PATH(e) into the cycle CYCLE(e). The set of cycles obtained this way is a depth-first cycle basis [10]. Let PATH(f) be the father of PATH(e) in PATR(G, T, r). From theorem 9 of [10], we see that SEG(e) is a segment of CYCLE(f) in the graph G. For any segment $S = \text{SEG}(e)$, we shall associate $\text{TAIL}(S) = \text{TAIL}(e)$ and $\text{LOW1}(S) = \text{LOW1}(e)$. These objects are the tail and low values of segments discussed in [10].

Given a node PATH(e) in PATR(G, T, r), we define the linearly ordered list of segments SEGLIST(e) as

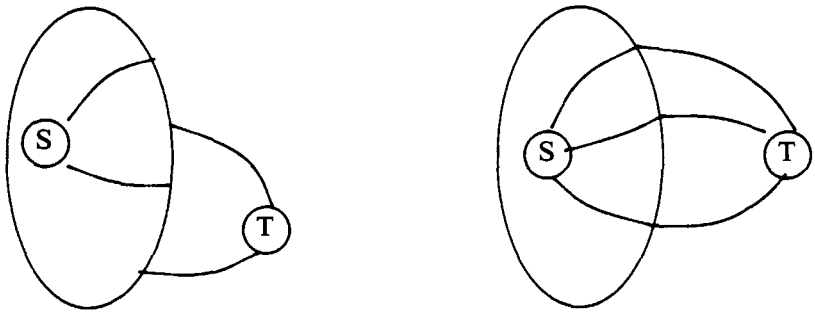
SEGLIST(e) = {SEG(x): PATH(x) is a son of PATH(e)}. The

order in SEGLIST(e) is the order in which the PATH(x)'s are generated during the construction of PATR(G, T, r). The following result is due to Hopcroft and Tarjan [6, 11].

LEMMA 3 *Let $X < Y$ be segments in* SEGLIST(e). *Then X, Y are directly linked relative to* CYCLE(e) *iff X has a backedge (s,t) so that* $\text{LOW1}(Y) < t < \text{TAIL}(Y)$.

Proof Since $X < Y$, by the ordering of edges, we must have $\text{TAIL}(X) \geqslant \text{TAIL}(Y)$. If $\text{TAIL}(X) > \text{TAIL}(Y)$, the condition that X has a backedge (s,t) with $\text{LOW1}(Y) < t < \text{TAIL}(Y)$ implies that X and Y cannot bridge each other and they cannot be p-disjoint. So they must be directly linked. On the other hand, if $\text{TAIL}(X) = \text{TAIL}(Y)$ we must also have $\text{LOW1}(X) = \text{LOW1}(Y)$. Now the above condition implies that Y also has a backedge (s', t') with $\text{LOW1}(X) < t' < \text{TAIL}(X)$. So X and Y are directly linked.

The following figure shows the two situations considered in the above proof.



LEMMA 4 *Let X, Y be segments in* SEGLIST(e). *If X bridges Y but Y does not bridge X then $X < Y$.*

Proof By the edge ordering, either $\text{TAIL}(X) > \text{TAIL}(Y)$ or $\text{TAIL}(X) = \text{TAIL}(Y)$ and $\text{LOW1}(X) < \text{LOW1}(Y)$.

Let PATH(e) be a vertex of PATR(G, T, r). We let SEGGR(e) be the graph with vertex set being SEGLIST(e) and edge set $\{(X, Y) : X$

d.1. Y}. This is the restricted segment graph considered in theorems 16 and 23 of [10]. As discussed in [10], the connected components of SEGGR(e) also have bridging, p-disjoint relations among themselves. We now have:

LEMMA 5 *Let C be a component of* SEGGR(e) *such that C does not bridge any other components. Then the set of segments that are nodes of C forms a contiguous block in* SEGLIST(e).

Proof This follows from lemmas 3, 4 above and lemma 4 of [10].

LEMMA 6 *Let M be a maximal set of segments in* SEGLIST(e) *that mutually bridge. The elements of M form a contiguous set of segments in* SEGLIST(e).

Proof This is a direct result of lemma 4.

The above lemmas are very important in constructing a linear time algorithm to detect separation pairs that correspond to TAIL and LOW1 of minimal components (minimal in the bridging order, see the proof of theorem 15 in [10]). Lemma 6 is useful for detecting multiple edges that arise from splittings.

DEFINITION 6 Let PATH(e) be a node in PATR(G, T, r). A segment S in SEGLIST(e) is said to be *internal* if S has a backedge (s, t) with LOW1(e) $< t <$ TAIL(e).

The following is proved in [10] (theorem 16):

THEOREM 7 *Let G be a simple biconnected graph without vertex of degree two. Let* PATR(G, T, r) *be a path tree for G. G is triconnected if*:

a. SEGGR(d) *is connected where* PATH(d) *is the root path of* PATR(G, T, r), *and*

b. *for every nonroot path* PATH(e), *every connected component of* SEGGR(e) *must contain at least one internal segment.*

V. DETECTION OF SEPARATION PAIRS AND DATA STRUCTURE

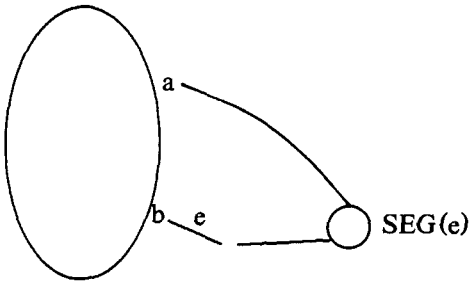
Although theorem 7 gives a good characterization of triconnectivity, the most straightforward algorithm that can be constructed using

this characterization would take at least quadratic time in the number of edges. This is because detecting the connectivity structure of segment graphs requires that much time in general. However, the theorem only says that each component of SEGGR(e) must contain an internal segment, it *does not* assert anything about the internal structure of a component. In the following, we shall discuss a method to detect the set of segments that belong to a component without actually constructing the connectivity structure of SEGGR(e). This can be done in linear time.

1. DETECTING SEPARATION PAIRS

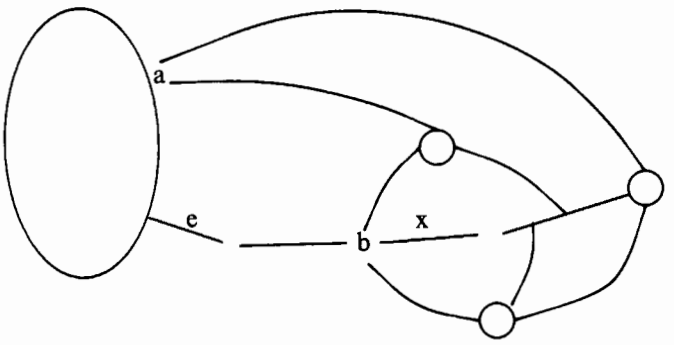
The separation pairs will be detected along with the building of the path tree in the recursive algorithm BUILD-PATR. Barring vertices of degree two, from the proof of theorem 7 (theorem 16 of [10]), we can divide the separation pairs $\{a,b\}$ of G into three types:

a. *Type 1 pair*: this corresponds to the TAIL and LOW1 points of a segment that has only two points in common with its parent cycle in the path tree. If this segment is SEG(e), then LOW2(e) = TAIL(e).

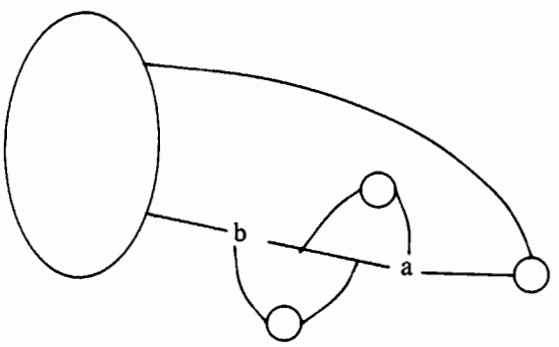


b. *Type 2a pair*: this is the LOW1 and LOW2 points of a component of SEGGR(e) with no internal segment as in the following picture. Let x be the tree edge on PATH(e) incident on the LOW2 point as in the picture, then LOW2(x) = TAIL(x). So this type of

separation pair can also be checked in the same way as the type 1 pairs.



c. *Type 2b pair*: this is TAIL and LOW1 points of a component without internal segments as shown in the picture. This type is the most difficult to detect.



Before discussing the detection of type 2b separation pairs, we need to define another object associated with vertices of G. We assume that the paths in PATR(G, T, r) are linearly ordered in the order that they are generated. To each vertex v, we associate a first path FIRSTP(v)

as the first generated path that has its backedge pointing into v. If v has no backedge pointing into it, FIRSTP(v) is defined to be ∞.

To detect type 2b pair, *local to each stage* in the basic recursive algorithm BUILD-PATR below we maintain a stack of pairs (LS, LOWC). LS is the earliest segment in SEGLIST(e) of a partially built component and LOWC is the current low point of this component. When a minimal component (in the bridging order) is detected, the corresponding separation pair is {LOWC, TAIL(LS)}. We emphasize the localness of the stack because every time we return from a recursive call, the stack on the stage that we are coming back from must be considered disappeared. Let PAIRS denote this local stack of pairs. PAIRS is updated as follows:

a. When a new segment SEG(x) in SEGLIST(e) is started, we pop from the stack PAIRS any pair (LS, LOWC) with LOWC > LOW1(x). If LOW1(x) $\geqslant$ TAIL(e) then: If (LS, LOWC) was the last pair deleted, put (LS, LOW1(x)) on the stack. If no pair was deleted, put (SEG(x), LOW1(x)) on the stack.

b. When we backtrack past an edge x on PATH(e), every pair (LS, LOWC) with LS > FIRSTP(TAIL(x)) is deleted.

The first type of update puts onto the stack PAIRS any possible type 2b pairs. By lemma 3, we know that SEG(x) is directly linked to some segment of the partial component corresponding to the deleted pair (LS, LOWC). So LOW1(x) is the current low point of this partially built component. The update makes sure that the earliest segment of the component is recorded correctly. Lemma 5 guarantees that if (LS, LOWC) corresponds to a minimal component, all segments of the component will be right on top of the partially built SEGLIST(e).

The second type of update removes pairs that do not define separation pairs at all. Let P = FIRSTP(TAIL(X)). Let C be the partial component corresponding to a deleted pair (LS, LOWC). The fact that $P <$ LS implies that P is descendant of some segment of a partial component C' with TAIL(C') $\geqslant$ TAIL(C) and LOW1(C') $\leqslant$ LOW1(C). This means that some segment of C is directly linked to the segment of C' that contains P. So C and C' is a part of a bigger component that can be identified by the pair corresponding to C'.

Finally, as we backtrack past an edge x in PATH(e), we check to see if the top pair (LS, LOWC) on PAIRS satisfies LOWC

$=$ TAIL(x). If this is the case, we have found a type 2b pair. The set of segments S on the partial SEGLIST(e) with $S \geqslant$ LS and the part of PATH(e) between TAIL(LS) and LOWC forms split component. Further, by construction this split component is triconnected.

The total number of pairs put onto the various stacks is proportional to the number of generated paths, so it is $O(V + E)$.

2. DATA STRUCTURES USED

For each vertex v, we associate a degree function DEG(v) that tells the number of edges incident on v. For each directed edge $e = (a, b)$, HEAD(e) $= b$, TAIL(e) $= a$.

Each node PATH(e) in the path tree PATR(G, T, r) will be associated with:

PATH(e).TAIL: the tail of the edge e,

PATH(e).LOW1: the low1 value of e,

PATH(e).EDGES: a doubly linked list [1] of edges of PATH(e),

PATH(e).SEGLIST: a doubly linked list representing the segment list,

PATH(e).ENUM: the total number of edges in SEG(e),

PATH(e).FATHER: the address of the edge a in the father path of PATH(e) that is incident to the tail of PATH(e). This is needed to facilitate splitting type 2b pairs.

We shall use two external integers. ETOTAL is the total number of edges remaining in the graph after each split. VIR is the number to be assigned to the next virtual edge if a split occurs.

As a shorthand, if a, b are numbers, we shall use:

$a \mathrel{+}= b$ for $a := a + b$ and

$a \mathrel{-}= b$ for $a := a - b$.

VI. THE SPLITTING ALGORITHM

This section describes the splitting algorithm for a biconnected simple graph $G = (V, E)$. We shall assume that G has been properly decomposed into $T^* + B^*$. Let d be the unique tree edge incident to the root of the lineal spanning tree (T, r). Following is the splitting

algorithm:

```
procedure SPLIT(G)
begin
  for v ∈ V do FIRSTP(v) := ∞;
  VIR := 1;
  ETOTAL := |E|;
  BUILD-PATR(d);
end
```

The algorithm to build the path tree, find separation pairs and actually split the graph follows:

```
 1) procedure BUILD-PATR(e)
 2) begin
 3)   PATH(e).TAIL := TAIL(e);
 4)   PATH(e).LOW1 := LOW1(e);
 5)   GETPATH(e, PATH(e));
 6)   if FIRSTP(LOW1(e) = ∞ then
      FIRSTP(LOW1(e)) := PATH(e);
 7)   for a ∈ PATH(e).EDGES (reading backward) do
 8)   begin
28)     add PATH(f) to PATH(e).SEGLIST;
 9)     t := TAIL(a); 12 := LOW2(a);
10)     while (LS, LOWC) satisfies LOWC = t do
11)     begin
12)       TYPE-2B(PATH(e), LS, LOWC);
13)       if DEG(LS.TAIL) = 2 and ETOTAL ⩾ 4 then
14)         TRIANGLE(PATH(e), a);
15)       delete (LS, LOWC) from the stack PAIRS;
16)     end
17)     if a is not a backedge then
18)       if DEG(HEAD(a)) = 2 and ETOTAL ⩾ 4 then
19)         TRIANGLE(PATH(e), a);
20)       else if 12 ⩾ t then TYPE-1-2A(PATH(e), a);
21)     delete from PAIRS any (LS, LOWC) with LS
        > FIRSTP(t);
22)     if a ≠ e then
23)     for f in T* + B*, f is unused and incident to t do
24)     begin
25)       BUILD-PATR(f);
```

```
26)          if PATH(f).ENUM = 1 and ETOTAL ⩾ 4 and
27)            CHECK-BOND(PATH(e), a, f) = TRUE then
               continue;
28)          add PATH(f) to PATH(e).SEGLIST;
29)          PATH(f).EFATHER := a;
30)          delete all (LS, LOWC) with LOWC > LOW1(f);
31)          if PATH(f).LOW1 ⩾ TAIL(e) then
32)            if (LS, LOWC) was the last pair deleted then
33)              add (LS, PATH(f).LOW1) to PAIRS;
34)            else add (PATH(f), PATH(f).LOW1) to PAIRS;
35)        end
36)      end
37)      for S ∈ PATH(e).SEGLIST do
         PATH(e).ENUM + = S.ENUM;
38)  end
```

Remarks

1: e is the tail edge of a new segment to be built.

3–5: set up information of the current path.

6: set up the first path parameter of the pertinent vertex.

7: backtracking along each edge of PATH(e) (see III.2).

9: it is important to save the tail and low1 values of a now because if a split occurs later that involves a, a will be gone. Of course there will be virtual edges in place of a, but we do not know their low point values.

10–16: check to see if there is a type 2b pair. If so split off the minimal component associated with it. Lines 13 and 14 take care of any degree two vertices resulting from the split.

17–20: check for degree two vertices and type 1, 2a pairs. If there is any, split off the corresponding split component.

21: this is the second type of update as discussed in V.1.

22–23: each edge f considered in these two lines spawns a new segment in SEGLIST(e).

25: the basic recursive call of BUILD-PATR.

26–27: if a segment has only one edge in it, it may form a multiple edge with some other one-edge segment that comes before it SEGLIST(e) or with some edge of PATH(e). If this happens, CHECK-BOND splits off the bond and return a value TRUE. The rest of the loop is skipped because there is no new segment to add to SEGLIST(e).

28–29: add PATH(f) as a new segment in SEGLIST(e). The edge EFATHER must be set so that splitting part of PATH(e) can be done easily during a type 2b split.

30–34: the first type of update as discussed in V.1.

37: find the exact number of edges in the segment SEG(e).

The total time spent in BUILD-PATR can be distributed to two factors, the splits and the updates of the PAIRS stacks. Using lemma 1, the time for all splits is linear in the number of edges. Updating the PAIRS stack is linear in the number of edges because the number of pairs added to the stacks is proportional to the number of backedges (or vertices of the path tree).

The algorithm for finding a path starting from some edge e follows.

```
procedure GETPATH(e, P)
begin
  P.EDGES := ∅;
  P.ENUM := 0;
  while TRUE do
  begin
    add e to P.EDGES and mark e as an used edge;
    P.ENUM += 1;
    if e is a backedge then return (end of path);
    e := first unused edge in T* + B* incident to HEAD(e);
  end
end
```

Following is the algorithm to split off a triangle when a vertex of degree two is found.

```
1)  procedure TRIANGLE(P, a)
2)  begin
3)    v := TAIL(a);
4)    b := the edge incident on v;
5)    w := HEAD(b);
6)    output a, b, (v, w, VIR) as a new split component;
7)    replace a, b by (v, w, VIR) in P.EDGES;
8)    P.ENUM −= 1;
9)    VIR += 1;
10)   ETOTAL −= 1;
11) end
```

Remarks

1: P is the path being considered, a is the edge with head having degree two.
3–6: split off the triangle.
7–8: update edge information for the path P.
9: increase the virtual edge counter.
10: update the total number of edges remaining in G.

The next algorithm checks if a bond exists. If so, it splits off the bond and returns a value TRUE. Note that there are three types of bonds that the edge f and form: a bond with the backedge of P, with a previous one-edge segment in P.SEGLIST, or with the tree edge of P.

```
1)  procedure CHECK-BOND(P, a, f)
2)  begin
3)    if f = the backedge of P then
4)      return BOND(P, b, f);
5)    if f = a then
6)      return BOND(P, a, f);
7)    S := the previous segment in P.SEGLIST;
8)    if S has only one edge e and e = f then
9)      return BOND(S, e, f);
10)   return FALSE;
11) end
12) procedure BOND(S, e, f)
13) begin
14)   g := (TAIL(e), HEAD(e, )VIR);
15)   output g, e, f as a new split component;
16)   change the edge e in S.EDGES to g;
17)   DEG(TAIL(e)) − = 1;
18)   DEG(HEAD(e)) − = 1;
19)   VIR + = 1;
20)   ETOTAL − = 1;
21)   return TRUE;
22) end
```

Remarks

1: P is the path being considered, a is the tree edge from which f stems out as a new segment.
3–4: check to see if f forms a bond with the backedge of P.

5–6: check to see if f forms a bond with a.

7–10: check to see if f forms a bond with a previous segment. Here lemma 6 guarantees that such a segment will be on top of the partially built P.SEGLIST.

12–22: the algorithm to actually split off the bond.

The next procedure splits off type 1 and 2a split components.

```
1)  procedure TYPE1-2A(P,a)
2)  begin
3)    for S ∈ P.SEGLIST do
4)    begin
5)      output every edge e in S and its descendants to
6)        the new split component, also subtract 1
7)        from DEG(TAIL(a))(DEG(HEAD(a)) if e is
8)        incident on TAIL(a)(HEAD(a));
9)      P.ENUM -= S.ENUM;
10)     ETOTAL -= S.ENUM;
11)   end
12)   for every edge f on P from a to the backedge of P do
13)   begin
14)     add f to the new split component;
15)     delete f from P.EDGES;
16)     P.ENUM -= 1;
17)     ETOTAL -= 1;
18)   end
19)   b := (TAIL(a), LOW1(a), VIR);
20)   add b to the new split component;
21)   add b as the new backedge of P.EDGES;
22)   P.ENUM += 1;
23)   ETOTAL += 1;
24)   VIR += 1;
25) end
```

Remarks

1: P is the path working on, a is the edge satisfying the conditions of types 1 and 2a pairs.

3–11: split off all subsegments of P. Line 3 is correct because if this is a type 2a, the new split component contains all segments on P.SEGLIST at this point. If this is a type 1 pair, of course, we split off everything on P.

12–18: if this is a type 2a pair, split off the part of P between the edge a and the backedge of P. If this is a type 1, all of P is split off.
19–24: add in the virtual edge.

Finally, we present the procedure to split off a type 2b pair.

```
1)  procedure TYPE-2B(P, LS, LOWC)
2)  begin
3)    for S ∈ P.SEGLIST and S ⩾ LS do
4)    begin
5)      output every edge e of S and its descendants to
6)        a new split component. At the same time
7)        subtract 1 from DEG(LOWC)(DEG(LS.TAIL))
8)        if e is incident on LOWC(LS.TAIL);
9)      remove S from P.SEGLIST;
10)     P.ENUM −= S.ENUM;
11)     ETOTAL −= S.ENUM;
12)   end
13)   for every edge f on P between LS.TAIL and LOWC do
14)   begin
15)     add f to the new split component;
16)     remove f from P.EDGES;
17)     P.ENUM −= 1;
18)     ETOTAL −= 1;
19)   end
20)   e := (LOWC, LS.TAIL, VIR);
21)   add e to the new split component;
22)   insert e into P.EDGES between LOWC and LS.TAIL;
23)   P.ENUM += 1;
24)   ETOTAL += 1;
25)   VIR += 1;
26) end
```

Remarks

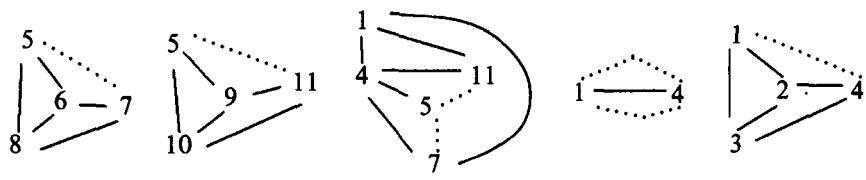
1: P is the path working on, LS is the address of the lowest segment in the component of the segment graph to be split off, and LOWC is the low point of this component.
3–12: split off all segments of the minimal component.
13–19: split off the edges on P that belong to the new split component. Note that the test that f is between LS.TAIL and LOWC

can be done easily only because we keep LS.EFATHER (recall that *P*.EDGES is a linked list).

20–25: add in the new virtual edge.

This completes our description of the splitting algorithm. The discussion after the presentation of BUILD-PATR shows that the time spent in BUILD-PATR is $O(|E|)$. It is easy to see that the time required to build $T^* + B^*$ is $O(|V| + |E|)$ (see [11]). Thus, the algorithm required $O(|V| + |E|)$ time.

Example 5 The split components of the graph G considered before are listed in the order that they are detected by the algorithm. The dash lines represent the virtual edges.



References

[1] A. V. Aho, J. Hopcroft, J. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

[2] J. Bruno, K. Steiglitz, L. Weinberg, *A New Planarity Test Based on Triconnectivity*, *Ibid.*, **17** (1970), pp. 197–206.

[3] S. Even, R. Tarjan, Network flow and testing graph connectivity, *SIAM J. Comp.*, **4** (1975), pp. 507–518.

[4] S. Even, Algorithm for determining whether the connectivity of a graph is at least *k*, *SIAM J. Comp.* **4** (1977), pp. 393–396.

[5] J. Hopcroft, R. Tarjan, *Isomorphism of Planar Graphs*, R. Miller, J. Thatcher, eds., *Complexity of Computer Computations*, Plenum Press, N.Y. (1972).

[6] J. Hopcroft, R. Tarjan, Dividing a graph into triconnected components, *SIAM J. Comp.*, **2** (1973) pp. 135–158.

[7] S. MacLane, A structural characterization of planar combinatorial graphs, *Duke Math. J.*, **3** (1937), pp. 460–472.

[8] R. Tarjan, Depth-first search and linear graph algorithms, *SIAM J. Comp.*, **1** (1972), pp. 146–159.

[9] W. Tutte, *Connectivity in Graphs*, U. of Toronto Press, 1966.

[10] K. P. Vo, *Segment Graphs, Depth-First Cycle Bases, 3-Connectivity and Planarity of Graphs*, Preprint, 1981.

[11] S. G. Williamson, Embedding graphs in the plane-algorithmic aspects, *Ann. Disc. Math.* **6** (1980), pp. 349–384.