Decomposing a Multigraph into Split Components

Yung H. Tsin\(^1\)

\(^1\) School of Computer Science
University of Windsor,
Windsor, Ontario, Canada,
Email: peter@uwindsor.ca

Abstract
A linear-time algorithm for decomposing a graph into split components is presented. The algorithm uses a new graph transformation technique to gradually transform the given graph so that every split component in it is transformed into a subgraph with very simple structure which can be easily identified. Once the split components are determined, the triconnected components of the graph are easily determined. The algorithm is conceptually simple and makes one less pass over the input graph than the existing best known algorithm which could mean substantial saving in actual execution time. The new graph transformation technique may be useful in other context.

Keywords: Graph algorithm, depth-first search, graph-connectivity, 3-vertex-connectivity, triconnectivity, triconnected component, separation pairs, split components.

1 Introduction
Graph connectivity is a basic property of graph that is fundamental to the studies of many other topics such as network reliability, graph drawing, quantum physics, bioinformatics and social networks. The notion of \(k\)-vertex-connectivity and \(k\)-edge-connectivity are two important concepts in graph connectivity. An undirected connected graph is \(k\)-vertex-connected (\(k\)-edge-connected, respectively) if disconnecting it requires the removal of at least \(k\) vertices (edges, respectively).

The \(1\)-vertex-connectivity and \(1\)-edge-connectivity problems are trivial. Tarjan (1972) presented the first linear-time algorithm for \(2\)-vertex-connectivity (also called biconnectivity). His algorithm is based on the graph traversal technique, depth-first search. The algorithm also solves the \(2\)-edge-connectivity problem in linear time. Gabow (2000) revisited depth-first search from a different perspective — the path-based view — and presented new linear-time algorithms for biconnectivity and \(2\)-edge-connectivity. For \(3\)-vertex-connectivity (also called triconnectivity), Hopcroft and Tarjan (1973) presented a linear-time algorithm that is also based on depth-first search. For \(3\)-edge-connectivity, a number of linear-time algorithms have been proposed (Galil et al. 1993, Nagamochi et al. 1992, Taoka et al. 1992, Tsin 2007, 2009). No linear-time algorithm is known for \(k > 3\).

Hopcroft and Tarjan (1973) presented the first linear-time algorithm for triconnectivity. Although elegant, the paper contains quite a number of minor but crucial errors which make the algorithm very hard to understand and implement correctly. Gutwenger and Mutzel (2001) did an excellent job in listing a number of such errors and explained how to correct them. Unfortunately, their explanation for some of the errors were brief and no detailed explanation on how to implement the corrected algorithm in linear time was given. The only way to find that out is to read the code of their implementation which is available in Gutwenger et al. (2001). Vo (1983) presented an algorithm which resembles that of Hopcroft et al. and gives almost no detail on implementation. Miller et al. (1992) and Fussell et al. (1993) presented parallel triconnectivity algorithms. Although these algorithms are serializable and run in linear time, they are much more complicated and obviously less efficient (in terms of actual run-time) then Hopcroft et al. as they are meant for the PRAMs. Recently, Saifullah and Ungor (2009) showed that triconnectivity can be reduced to \(3\)-edge-connectivity in linear time, making it possible to use \(3\)-edge-connectivity algorithm to solve the triconnectivity problem.

In this paper, we present a conceptually simple linear-time algorithm that uses a new graph transformation technique to solve the triconnectivity problem. The idea is similar to that of Tsin (2007) for solving the \(3\)-edge-connectivity problem. In contrast with Hopcroft et al., our algorithm does not classify separation pairs into different types. Moreover, our algorithm avoids the rather time-consuming construction of a special adjacent lists structure for the input graph, hence making one less pass over the graph. Here, one pass is a process that requires examining the entire adjacency lists structure representing the graph, such as performing a depth-first search over the graph or sorting the edges of the graph with bucket sort. The graph-transformation technique could be useful in other context.

2 Definitions
The definitions of most of the graph-theoretic concepts used in this paper are standard and can be found in Even (1979) or Hopcroft and Tarjan (1973). We shall only give some important or uncommon definitions below.

A graph \(G = (V, E)\) consists of a set of vertices \(V\) and a set of edges \(E\). The graph is either undirected or directed. The graph is a multigraph if it contains parallel edges (edges with the same end-vertices) and is a simple graph otherwise. Let \(U \subseteq V\). The sub-
The graph of $G$ induced by $U$, denoted by $\langle U \rangle$, is the maximal subgraph of $G$ whose vertex set is $U$. The graph resulting from $G$ after all the vertices in $U$ are removed is denoted by $G - U$. A connected graph $G = (V, E)$ is biconnected if for all $v \in V, G - \{v\}$ is a connected graph.

Let $G = (V, E)$ be a biconnected graph and $a, b \in V$. \forall e, e' \in E$, let $e \sim e'$ if and only if there exists a path containing both $e$ and $e'$ but not $a$ or $b$ except as a terminating vertex. The binary relation $\sim_{(a,b)}$ is an equivalence relation in $E$. Therefore, the edge set $E$ can be partitioned into equivalence classes $E_1, E_2, \ldots, E_k$ w.r.t. $\sim_{(a,b)}$. If $k \geq 2$, then $(a, b)$ is a separation pair unless $(i) k = 2$ and $\exists i \in \{1, 2\}, E_i = \{(a, b)\}$, or (ii) $k = 3$ and $E_i = \{(a, b)\}, 1 \leq i \leq 3$. A biconnected graph is triconnected if it has no separation pair.

Let $(a, b)$ be a separation pair and $E_a, 1 \leq i \leq k$, be the equivalence classes w.r.t. $\sim_{(a,b)}$. Let $E' = \bigcup_{i=1}^{k} E_i$ and $E'' = \bigcup_{i=1}^{k} E_i$, such that $|E'|, |E''| \geq 2$. Let $G_1 = (V_1, E' \cup \{e\})$ and $G_2 = (V_2, E'' \cup \{e\})$ such that $e = (a, b)$ and $V_1$ and $V_2$ (respectively) consists of the end-vertices of the edges in $E'$ (respectively). The graphs $G_1$ and $G_2$ are called split graphs of $G$ w.r.t. $(a, b)$. Replacing $G$ by $G_1$ and $G_2$ is called splitting $G$. The new edge $e$ that is added to both $G_1$ and $G_2$ is called a virtual edge. The virtual edge identifies the split creating it. It is easily verified that if $G$ is biconnected, then any split graph of $G$ is also biconnected. Let $G$ be split into two split graphs, the splits graph are then split into smaller split graphs, and so on, until no more splits are possible. Then each of the resulting split graphs is called a split component. A split component is one of the following three types: triple bond (a graph consisting of two vertices and three parallel edges), triangle (a cycle of length three) and triconnected simple graph.

Let $G$ be a biconnected graph whose set of split components is $B_0 \cup T \cup \mathcal{G}$, where $B_0$ is a set of triple bonds, $T$ is a set of triangles, and $\mathcal{G}$ is a set of triconnected simple graphs. Let $\mathcal{B}$ be the set of multiple bonds obtained from $B_0$ by merging triple/multiple bonds that have common edges until no more merging is possible. Similarly, let $\mathcal{P}$ be the set of polygons obtained from $T$ by merging triangles/polynomials that have common edges until no more merging is possible. Then $B \cup \mathcal{P} \cup \mathcal{G}$ is the set of all triconnected components of $G$.

The following are some important facts about depth-first search: the search assigns every vertex $v$ a depth-first search number, $dfs(v)$, and converts $G$ into a directed graph, $P_G$, called a palm tree of $G$. An edges in $P_G$ is either a tree-edge, denoted by $(u \to v)$, or a frond, denoted by $(u \leftarrow v)$, where $u$ is the tail and $v$ is the head. The frond $(u \leftarrow v)$ is an incoming frond of $v$ and an outgoing frond of $u$. The tree edges form a rooted spanning tree, $T$, of $G$. A subtree of $T$ rooted at vertex $w$, denoted by $T_w$, is the maximal subgraph of $T$ that is a tree and of which $w$ is the root. The vertex set of $T_w$ is denoted by $V_{T_w}$ and the set of edges of $T_w$ is denoted by $E_{T_w}$ and $u \leftarrow v$ is called the current vertex of the search.

A millipede, denoted by $\widetilde{T}_k e_1 \widetilde{T}_1 e_2 \widetilde{T}_2 \cdots \widetilde{T}_k$, is a graph consisting of a path $u_0 e_1 u_2 e_2 u_3 \cdots e_k u_k$, and a set of trees, $\widetilde{T}_i, 0 \leq i \leq k$, such that each $\widetilde{T}_i$ is of height at most 1 and is rooted at $u_i$ (Figure 1).

The path is the spine while the edges in the trees are the legs of the millipede. Vertices $u_0$ and $u_k$ are the terminating vertices of the millipede. When $\widetilde{T}_0$ is a null tree, the millipede can be denoted by $e_1 \widetilde{T}_1 e_2 \widetilde{T}_2 \cdots \widetilde{T}_k$. We are interested in millipedes in which every edge is a superedge. A superedge $e$ is an edge associated with a set of edges, denoted by $\widetilde{e}$, that are tree edges or fronds of the palm tree $P_G$. A supergraph $G$ is a graph whose edges are superedges. An outgoing frond of a superedge $e$ is a frond $(x \leftarrow y)$ such that $x$ is an end-vertex of some edge in $\widetilde{e}$ but not of $e$ while $y$ is not an end-vertex of any edge in $\widetilde{e}$. An outgoing frond of a millipede is an outgoing frond of a superedge or of a vertex (excluding the two terminating vertices) in the millipede. The set of all outgoing fronds of a vertex $u$, of a superedge $e$ and of a millipede $\mathcal{P}$ are denoted by $Out(u)$, $Out(e)$ and $Out(\mathcal{P})$, respectively. An edge (as oppose to a superedge) of a millipede is an edge of $G$ that is either an edge in a superedge or an outgoing frond of the millipede.

Initially, the palm tree $P_G$ created by a depth-first search can be regarded as a supergraph in which $\widetilde{e} = \{e\}, \forall e \in E$. During the execution of our algorithm, $P_G = (P_G^1, P_G^2) = (P_G^1, P_G^2) \cdots$ is transformed gradually into a sequence of supergraphs $P_G^1, P_G^2, \ldots, P_G^q$ such that every split component of $P_G^q$ corresponds to a split component of the input graph $G$ and vice versa. Specifically, $P_G^j, 0 \leq j < q$, is transformed into $P_G^{j+1}$ either by splitting it into two or three subgraphs, or by having two superedges coalesced into one superedge. Both transformations occur on a millipede in $P_G^j$. Notice that even though $P_G$ is a directed graph, we shall regard it as the undirected graph $G$ in which every edge is classified as either a tree-edge or a frond. As a result, all of the definitions for undirected graph given above apply equally well to $P_G$ as well as $P_G^q, 1 \leq j \leq q$. However, as the tails of the outgoing fronds of the superedges are not vertices in $P_G^q$, the definition of the equivalence relation, $\sim_{(a,b)}$, in $P_G^j$ must be modified accordingly: $e \sim_{(a,b)} e'$ if and only if there is a path for which each of $e$ and $e'$ is either an edge on it or an outgoing frond of a superedge on it and neither $a$ nor $b$ is an internal vertex. Similarly, the definition of path is modified as follows: a path in $P_G^j, 0 \leq j \leq q$, is a sequence of edges $e_1 e_2 \ldots e_q$ such that either $e_i$ and $e_{i+1}$ share a common end-vertex or one of them is an outgoing frond of the other. The following terms were first introduced in (Hopcroft and Tarjan 1973).
3 The triconnectivity algorithm

Given an undirected multigraph $G = (V, E)$. If $G$ contains parallel edges, we can follow Hopcroft and Tarjan (1973) to separate them, creating a set of trile edges and a simple graph $G'$ without parallel edges. If $G'$ is not biconnected, we can use the biconnectivity algorithm of Tarjan (1972) to decompose $G'$ into a collection of biconnected components. Hence, in the following discussion, we shall assume without loss of generality that the graph $G$ is biconnected and simple.

Since $G$ is biconnected, it is easily verified that if $\{x, y\}$ is a separation pair, then one of the two vertices is an ancestor of the other in the depth-first search tree (see Hopcroft and Tarjan (1973)). The basic idea underlying our algorithm is as follows. In Figure 2(a) and (b), it is obvious that the vertex pair $\{a, b\}$ is a separation pair and the triangle $ae_1webe'a$, where $e'$ is a virtual edge, is a split component. Clearly, not every split component is of that simple structure. Our algorithm will transform the graph during a depth-first search gradually so that every split component is transformed into a millipede whose spine consists of two or more superedges (the $a - b$ path in Figure 3(a), (b)) such that no outgoing edge of the superedges or of the internal vertices on the millipede has its other end-vertex outside the millipede. This condition will be detected when the search backtracks to one of the end-vertices (vertex $a$ in Figure 3(a), (b)). A split component will then be created.

Two transformation operations will be used by our algorithm to transform the given graph $G$ (or more precisely the palm tree $P_G$) gradually into a collection of split components. The first one is called split which will be used to separate a millipede from the supergraph to which it belongs so as to produce a split component.

The second one is called coalesce which is to be applied to a millipede whose spine consists of two superedges after the internal vertex of the spine has been confirmed to be unable to form new separation pairs. Specifically, let $e_1T_1e_2$ be a millipede in which the spine is $u_0u_1u_2$, where $e_1 = (u_0, u_1)$ and $e_2 = (u_1, u_2)$. When a coalesce operation is applied to the millipede, the millipede is replaced by a new superedge $e_1' = (u_0, u_2)$ such that $e_1'$ consists of the edges of the superedges in the millipede while $Out(e_1')$ consists of the outgoing edges of the millipede (if an outgoing edge becomes an internal edge of $e_1'$, it is included in $e_1'$ rather than $Out(e_1)$). The coalesce operation can be easily extended to millipedes having more that two superedges on its spine.

The following is a brief description of the algorithm. For ease of explanation, we shall use $low(u)$ and $y$ interchangeably if $dfs(y) = low_1(u)$.

First, starting from an arbitrary vertex $v$, a depth-first search is performed over the graph $G$ to construct a palm tree $P_G$ of $G$. During the search, at each vertex $u$, $low_1(u)$ and $low_2(u)$ are computed. If there is a child $v$ of $u$ such that $low_1(v) = low_1(u)$, then among all these children, one of them is made the first child of $u$. If there is no such a child $v$ of $u$, then an outgoing frond of $u$, $u \leftarrow y$, such that $dfs(y) = low_1(u)$ is made the first frond of $u$. A first descendant of vertex $u$ is the first child of $u$ of a first descendant of the first child of $u$.

A second depth-first search is then performed over $P_G$. Recall that $P_G$ can be regarded as the graph $G$ with every edge in it being classified as tree-edge or frond.

During this search, at each vertex $u$, the first incident edge traversed is the one connecting the first child or the first frond if the first child does not exist. When the search backtracks from a vertex $u$ to the parent vertex $w$, the subgraph of $G$ consisting of the edge set of $(T_u)$ and the fronds that have an end-vertex in $T_u$ has been transformed into a graph consisting of a set of split components and a millipede $\hat{P}_u = \hat{T}_0\hat{e}_1\hat{T}_1 \cdots \hat{e}_k\hat{T}_k$ called the u-millipede, with the following properties: (Figure 4(i))

(i) the spine of the millipede $u_0u_1 \ldots u_k$ is such that $u_0 = w; u_{j+1}$ is a first descendant of $u_j, 0 \leq i \leq k,$ and $u_k$ has an outgoing frond $f = (u_k \leftarrow y)$ such that $dfs(y) = low_1(u_k)$. Furthermore, if $k > 0$, then $e_k\hat{T}_k$ or vertex $u_k$ has an outgoing frond $f' = (x' \leftarrow y')$ such that $y'$ is an internal vertex of the path $low_1(u) \leftarrow u$.

(ii) for every superedge $e$ in $\hat{P}$, $\hat{e}$ consists of all the edges that have an end-vertex in $T_u$ and are known to be belonging to the same split component as $e$.

(iii) for every outgoing frond $f = (x \leftarrow y)$ of $\hat{P}$ (including the terminating vertex $u_k$), either $dfs(y) < dfs(u)$ or $f = (u_i \leftarrow u_{i-1})$, for some $i, 1 \leq i \leq k$.

Let $\tilde{P} : e_0\tilde{T}_0\tilde{e}_1\tilde{T}_1 \cdots \tilde{e}_k\tilde{T}_k$ be the millipede resulting from concatenating the superedge $e_0 = (w \leftarrow u)$
the resulting millipede be until the aforementioned condition does not hold. Let

\[ P \]

is replaced by a virtual edge. This process is repeated going frond applies to the section

\[ P \]

with \( \tilde{P}_w \). If there is no outgoing frond of \( e_0 \tilde{T}_0 e_1 \) whose head is outside \( e_0 \tilde{T}_0 e_1 \), then \( \{ w, u_1 \} \) is a separation pair and the edges in \( e_0 \tilde{T}_0 e_1 \) form a split component (this corresponds to the situation depicted in Figure 3(a)). The section \( e_0 \tilde{T}_0 e_1 \) on \( \tilde{P} \) is then replaced by a virtual edge \( e'_1 = (w, u_1) \). If there is a frond \( f' = (u_1 \rightarrow w) \), then a triple bond with vertex set \( \{ w, u_1 \} \) is also created. Let the updated \( \tilde{P} \) be \( e'_1 \tilde{T}_1 e_2 \tilde{T}_2 \cdots e_k \tilde{T}_k \). If the above condition on outgoing frond applies to the section \( e'_1 \tilde{T}_1 e_2 \), then a split component is generated and the section \( e'_1 \tilde{T}_1 e_2 \) on \( \tilde{P} \) is replaced by a virtual edge. This process is repeated until the aforementioned condition does not hold. Let the resulting millipede be \( e'_1 \tilde{T}_1 e_2 \tilde{T}_2 \cdots e_k \tilde{T}_k \). Let \( \tilde{P} \) be \( e'_1 \tilde{T}_1 e_2 \tilde{T}_2 \cdots e_k \tilde{T}_k f \) such that \( f = (u_k \rightarrow y) \), where \( df_s(y) = low1(u_k) \). If there is no outgoing fronds of \( \tilde{P} \) whose head is an internal vertex of the tree-path \( low1(u_k) \rightarrow w \) and there is at least one vertex outside \( V_{T_{u_k}} \cup \{(w, low1(u_k))\} \), then \( \{ w, low1(u_k) \} \) is a separation pair and the entire millipede \( \tilde{P} \) is removed to produce a split component (Figure 4(iii)) (this corresponds to the situation depicted in Figure 3(b)). A virtual frond \( e' = (w \leftarrow low1(u_k)) \) is introduced to replace \( \tilde{P} \). If there is already a frond \( f' = (w \leftarrow low1(u_k)) \) in \( P_G \), then a triple bond with vertex set \( \{ w, low1(u_k) \} \) is also created. On the other hand, if there is an outgoing fronds of \( \tilde{P} \) whose head is an internal vertex of \( low1(u_k) \rightarrow w \), then \( \tilde{P} \) is coalesced into a superedge \( e'_k = (w, u_k) \) if \( u_k \) is not a first descendant of \( w \). Otherwise, \( \tilde{P} \) becomes the \( w \)-millipede \( \tilde{P}_w \).

After all the children of \( w \) are processed, let \( \tilde{P}_w \) be \( e'_1 \tilde{T}_1 e_2 \tilde{T}_2 \cdots e_k \tilde{T}_k \). The incoming fronds of \( w \) are examined. For each incoming frond \( f = (x \rightarrow w) \), if \( f \in Out(u_i) \cup Out(e_j) \), \( h < j \leq k \), then the vertices \( u_i, h \leq i < j \), can no longer generate separation pair. The section on \( \tilde{P}_w, e'_1 \tilde{T}_h e_{h+1} \tilde{T}_{j-1} e_j \), is thus coalesced into a superedge \( e'_j = (w, u_j) \). Similarly, if \( f \) is an outgoing frond of a leg \( e \) in some \( \tilde{T}_h, h \leq j \leq k \), then the section \( e'_1 \tilde{T}_h e_{h+1} \cdots e_j \) and the leg \( e \) are coalesced into a superedge \( e'_j = (w, u_j) \).

When the incoming fronds of \( w \) are completely processed, the \( w \)-millipede is finalized. The search then backtracks to the parent vertex of \( w \) unless \( w = r \); in which case, execution of the algorithm terminates and the splits components are all generated.

The palm tree \( P_T \) is represented by adjacency lists \( A[w], w \in V \), which are created during the first depth-first search. Specifically, an entry \( u \) in \( A[w] \) represents a tree-edge, \( (w \rightarrow u) \), if \( df_s(u) > df_s(w) \) and represents an outgoing frond of \( w, (w \rightarrow u) \), if \( df_s(u) < df_s(w) \). The first entry in \( A[w] \) is the first child of \( w \) if \( w \) has a first child and is the first frond of \( w \), otherwise. Note that the adjacency lists structure used in Hopcroft and Tarjan (1973) as it does not require the vertices in each list to be arranged in a particular order. The bucket-sort is thus avoided. Since this step is rather straightforward, we shall omit the details.
The second depth-first search is performed over $G$ based on the adjacency lists $A[w], \forall w \in V$. The split components are determined during this search. The details are presented as Algorithm Split-component below. In the algorithm, $w \oplus \tilde{P}$ denotes the millipede $e_1e_2\ldots e_k\tilde{T}_k$ if $e_1 = (w \rightarrow u_1)$ and $\tilde{P}$ is the millipede $\tilde{T}_1e_2\tilde{T}_2\ldots e_k\tilde{T}_k$. Note that to make the description of the second search (which is the main part of our algorithm) self-contained, we have included the calculations of $\text{low}(w), \text{low}(v), \forall v \in V$, even though they are calculated during the first search. Moreover, to efficiently determine if a millipede of the form $e_1\tilde{T}_1e_{i+1}$ produces a split component, the following terms are determined during this search.

$$\text{low}(w) := \min\{\text{low}(v), \text{dfs}(v)\}$$

$$\text{low}(v) := \min\{\text{low}(w), \text{dfs}(w)\}$$

**Procedure SplitComp**

Input: The adjacency lists $A[w], \forall w \in V$.
Output: The split components of $G = (V, E)$.

1. for each $(u \in A[w])$

   if $(e = (w \rightarrow u) \in E_T)$ then /* a tree edge */
     $e := \{e\};$ \text{low}(e) := \text{dfs}(w);
   else if $(u \notin \mu)$ then /* is the tail of an outgoing frond */
     $\mu := \{(u, u_{i+1})\};$
   else /* $e$ is an incoming frond */
     $\mu := \{(u, u_{i+1})\};$
     $i := i+1$;
end;

2. for each $(w \in A(w))$

   if $(e = (w \rightarrow u) \in E_T)$ then /* a tree edge */
     $e := \{e\};$ \text{low}(e) := \text{dfs}(w);
   else /* $e$ is an incoming frond */
     $\mu := \{(u, u_{i+1})\};$
     $i := i+1$;
end;

3. if $(\exists f = (u_{i+1} \rightarrow u_0))$

   if $(e_i = e_{i+1}, e_{i+1})$ then output(split component:
     $(e | e$ is an edge of $e_i\tilde{T}_ie_{i+1})$)
   else /* new virtual edge */
     $e_i := e_{i+1};$
   end;
end;

4. if $(\exists f = (u_{i+1} \rightarrow u_0))$

   if $(e_i = e_{i+1}, e_{i+1})$ then output(split component:
     $(e | e$ is an edge of $e_i\tilde{T}_ie_{i+1})$)
   else /* new virtual edge */
     $e_i := e_{i+1};$
   end;
end;

5. if $(\exists f = (u_{i+1} \rightarrow u_0))$

   if $(e_i = e_{i+1}, e_{i+1})$ then output(split component:
     $(e | e$ is an edge of $e_i\tilde{T}_ie_{i+1})$)
   else /* new virtual edge */
     $e_i := e_{i+1};$
   end;
end;
output( split component):
\{e | e is an edge in \( \tilde{P} \) : \( e_1 T_ne_{i+1} \ldots e_k T_k \) \} \cup
Out(u_k) \cup \{e_0\},
where
\( e_0 = (u_0 \rightarrow low1(u_0)) \) is a new virtual edge;
2. if \((3f = (u_0, low1(u_1)))\) then
output( triple bond: \( \{f, e_0, e_0'\}\)), where \( e_0' = (u_0, low1(u_1)) \) is a new virtual edge;
replace \( \tilde{P} \) with the null tree \( \tilde{T}_0 \) with \( u_0 \) as the root;
label rel \( f \) as \( e_0' \);
else replace \( \tilde{P} \) with the null tree \( T_0 \) with \( u_0 \) as the root;
Out(u_0) = Out(u_0) \cup \{e_0\};
low3(u_0) := min\{low3(u_0), low1(u_1)\};
return(\( \tilde{P} \));
end;

Lemma 3.1 Let \( w \in V - \{r\} \). During the depth-first search over \( P_G \), when the search backtracks from \( w \) to its parent vertex \( v \), the subgraph of \( P_G \) consisting of the edge set of \( (V_T) \) and the fronds that have an end-vertex in \( T_w \) has been transformed into a set of isolated millipedes and a \( w \)-millipede, \( P_w : \)
\( \tilde{T}_0 e_1 \tilde{T}_1 \ldots e_k \tilde{T}_k \), such that:
(i) \( u_0 = w \); \( u_{i+1} \) is a first descendant of \( u_i \), \( 0 \leq i < k \); \( 3f = (u_k \leftarrow y) \) such that df(s(y)) = low1(w),
and if \( k > 0 \), then \( 3f' = (x' \leftarrow y') \) such that \( x' \in Out(e_i \tilde{T}_k) \cup Out(u_k) \) and low1(w) < df(s(y')) < df(s(w));
(ii) \( \forall f = (x \leftarrow y) \in Out(\tilde{P}_w) \cup Out(u_k) \), either
\( df(s(y)) < df(s(w)) \) or \( f = (u_i \leftarrow u_{i-1}) \), for some \( i, 1 \leq i \leq k \).

Proof: (By induction on the height of \( w \) in \( T \))
The base case where \( w \) is a leaf is obvious.
Suppose the lemma holds true for every vertex with height \( < h \) (\( h \geq 1 \)).
Let \( w \) be a vertex with height \( h \) and \( u \) be a child of \( w \). Since the height of \( u \) is less than \( h \), by the induction hypothesis, when the depth-first search backtracks from \( u \) to \( w \), the subgraph of \( P_G \) consisting of the edge set of \( (V_T) \) and the fronds that have an end-vertex in \( T_u \) has been transformed into a set of isolated millipedes and a \( u \)-millipede satisfying conditions (i) and (ii). Let the \( u \)-millipede be \( P_u : \tilde{T}_0 e_1 \tilde{T}_2 \ldots e_k \tilde{T}_k \). Then Procedure Gen_SplitComponent is called to process the millipede \( w \oplus P_u \) which is \( \tilde{P} : e_1 \tilde{T}_1 e_2 \ldots e_k \tilde{T}_k \), where \( e_1 = (u_0, u_1) = (w, u) \).
Within Procedure Gen_SplitComponent, the while loop is repeated to produce isolated millipedes until \( \tilde{P} \) is reduced to the millipede \( e_1 \tilde{T}_h e_{h+1} \ldots e_k \tilde{T}_k \), where either \( h = k \) (i.e. \( \tilde{P} : e_1 \tilde{T}_k \)) or \( h < k \) such that min\{low3(u_h), low3(e_{h+1})\} < df(s(w)). If the condition in the if statement following the while loop holds for \( \tilde{P} \), then \( \tilde{P} \) is further reduced to the null path and an isolated millipede is generated based on it.
Now, if \( u \) is not the first child of \( w \), then Procedure Coalesce is invoked to reduce \( \tilde{P} \) to a supereude \( e = (w, u) \) (if \( \tilde{P} \) has not become the null path) which then becomes \( \tilde{P}_w \) in \( \tilde{T}_0 \).
On the other hand, if \( u \) is the first child of \( w \), then \( \tilde{P} \) becomes \( \tilde{P}_w \). If \( \tilde{P}_w \) has been reduced to a null path, it clearly satisfies the two conditions.

Suppose \( \tilde{P}_w \) is not a null path. After the adjacency list of \( w \) is completely processed, The incoming fronds of \( w \) are then processed.

Let \( \tilde{P}_w : e_1 \tilde{T}_h e_{h+1} \ldots \tilde{T}_k e_k \tilde{T}_k \) and \( e = (u \leftarrow w) \) be an incoming frond of \( w \). If \( u \) is not a descendant of \( u_h \), then \( e \) is in a leg in \( \tilde{T}_0 \) (rooted at \( w \)) or in a split component that was generated earlier. It is thus irrelevant. Otherwise, \( \exists l, h \leq l \leq k \), such that \( e \in Out(u_l) \cup Out(e_l) \cup Out(e') \), where \( e' \) is a leg in \( \tilde{T}_l \) and the section of \( \tilde{P}_w, e_1 \tilde{T}_h e_{h+1} \ldots \tilde{T}_l e_l \), is coalesced into a supereude. Moreover, if \( e \in Out(e') \), the supereude \( e' \) is coalesced with the aforementioned section of \( \tilde{P}_w \). When all the incoming fronds of \( w \) are processed, \( \tilde{P}_w \) must have been reduced to \( \tilde{T}_0 e_1 \tilde{T}_h e_{h+1} \ldots e_k \tilde{T}_k \), where \( h \leq m \leq k \) and \( e_1 = (w, u_m) \) is a new supereude such that \( \forall f = (x \leftarrow y) \in Out(\tilde{P}_w) \cup Out(u_k) \), if \( f \in Out(e_1) \cup Out(u_m) \), then \( df(s(y)) < df(s(w)) \) or \( f = (u_m \leftarrow w) \), and if \( f \in Out(\tilde{T}_m e_{m+1} \ldots e_k \tilde{T}_k) \cup Out(u_k) \), then by the induction hypothesis and the definition of \( u_m \), either \( df(s(y)) < df(s(w)) \) or \( f = (u_i \leftarrow u_{i-1}) \), for some \( i, m < i \leq k \). Condition (ii) thus holds for \( \tilde{P}_w \).

Since \( \tilde{P}_w \) is created from \( \tilde{P}_u \), where \( u \) is the first child of \( w \), by the induction hypothesis and the definition of \( u_m \), it is easily verified that Conditions (i) holds for \( \tilde{P}_w \). The lemma thus follows.

The following lemmas and corollary show that both the split and coalesce operations preserve bi-connectivity.

Lemma 3.2 Let \( P_G \) be transformed into a set of isolated millipedes and a supergraph \( P' \) before a split operation is applied to a pair of vertices \( \{a, b\} \) of \( P_G \) in Procedure Gen_SplitComp. Let \( P_G' \) be split into an isolated millipede \( C \) and a supergraph \( P'' \) after the split operation is applied. Then \( P_G' \) is biconnected implies \( P'' \) is biconnected.

Proof: Trivial.

Lemma 3.3 Let \( P_G \) be transformed into a set of isolated millipedes and a supergraph \( P_G' \) before a coalesce operation is applied at a vertex \( w \) of \( P_G \). Let \( P_G' \) be transformed into \( P_G'' \) after the coalesce operation is applied. Then \( P_G'' \) is biconnected implies \( P_G'' \) is biconnected.

Proof: Omitted

Corollary 1 During an execution of Algorithm Split-components, let \( P_G \) be transformed into a set of isolated millipedes and a supergraph \( P_G'' \) containing the root \( r \). \( P_G \) is biconnected implies \( P_G'' \) is biconnected.

Proof: Immediate from Lemmas 3.2 and 3.3.

The following three lemmas give simple criteria (based on the terms low2 and low3) for detecting split components using the idea illustrated in Figure 3.

Lemma 3.4 Suppose the depth-first search has backtracked from a vertex \( w \) to its parent vertex \( v \). Let the millipede \( v \oplus \tilde{P}_w \) be \( \tilde{T}_0 e_1 \tilde{T}_h e_{h+1} \ldots e_k \tilde{T}_k \) (\( 1 \leq h \leq k \)) at that point of time.

(i) \( \exists f = (x \leftarrow y) \) such that \( f \in Out(e_1 \tilde{T}_h e_{h+1}) \) and \( df(s(y)) < df(s(v)) \) if and only if the millipede \( e_1 \tilde{T}_h e_{h+1} \) forms a split component with \( \{v, u_{h+1}\} \) as the corresponding separation pair;
The following lemma provides the justification.

**Lemma 3.8** Let $P_G$ be transformed into a set of isolated millipedes and a supergraph $P'_G$ before a coalesce operation is applied at one of the vertices $w$ in $P_G$. Let $w \rightarrow u$ be the first edge of the millipede to which the coalesce operation is applied and $P'_G$ be the resulting supergraph. If no vertex in $P'_G$ can generate a separation pair with $u$, then $S$ is a set of separation pairs of $P'_G$ if and only if $S \cup \{a,b\}$ is a set of separation pairs of $P_G$. Moreover, for the collection of split components generated by $S$, $P'_G = P_G \cup \{M\}$.

**Proof:** Trivial.

**Lemma 3.9** Let $P_G$ be transformed into a set of isolated millipedes and a supergraph $P'_G$ before a coalesce operation is applied at one of the vertices $w$ in $P_G$. Let $w \rightarrow u$ be the first edge of the millipede to which the coalesce operation is applied and $P'_G$ be the resulting supergraph. If no vertex in $P'_G$ can generate a separation pair with $u$, then $S$ is a set of separation pairs of $P'_G$ if and only if $S \cup \{a,b\}$ is a set of separation pairs of $P_G$. Moreover, for the collection of split components generated by $S$, $P'_G = P_G \cup \{M\}$.

**Proof:** Trivial.

**Theorem 3.10** Algorithm Split-components decomposes a biconnected graph $G$ into split components.

**Proof:** By induction on the height of the vertices in $T$ based on Lemmas 3.1, 3.4, 3.5, 3.6, 3.7, 3.8, 3.9.

### 4 Time complexity

The first depth-first search creates a palm tree $P_G$ of $G$ and constructs an adjacency lists structure for it. It is easily verified that the adjacency lists can be constructed in $O(|V| + |E|)$ time. The second depth-first search is performed over the palm tree $P_G$ by invoking Algorithm Split-components. To ensure that this search runs in $O(|V| + |E|)$ time, the following issues must be addressed.

1. Performing the coalesce operation:
   - A millipede $T_v e_1, T_1, \ldots, e_k T_k$ is represented by a linked list $u_0 \rightarrow u_1 \ldots u_k$ (representing its spine) augmented by the following data structures:
     - $p(v)$, $\forall v \in V$: the parent of $v$.
     - $e_v$, $\forall v \in V - \{r\}$, where $e_v = (p(v) \rightarrow v)$. Since the parent edge of $v$ is unique, $e_v$ is determined by $v$; $e_v$ is thus attached to vertex $v$ so that retrieving it can be done in $O(1)$ time.
   - Linked lists are used to represent $Out(v), v \in V$, and $e_v, v \in V - \{r\}$, so that coalescing two superedges can be done in $O(1)$ time. In particular, in coalescing
a millipede \(e_1 \tilde{T}_1 \ldots e_h \tilde{T}_h\) into a superedge, where \(e_1 = (u_0, u_1), e_{i+1} = (u_{i}, u_{i+1})\); we let \(\tilde{e}_{i+1} := \tilde{e}_i \cup \tilde{e}_{i+1} \cup \bigcup_{e \in E_{\tilde{T}_i}} e \cup \text{Out}(u_i)\), i.e. we let the new superedge be the parent edge of \(u_{i+1}\).

When a frond \(f = (x \leftarrow u)\) is retrieved from \(\text{InFrondList}(u)\), the section of the \(u\)-millipede from \(u\) to \(u_i\) is to be coalesced into a single superedge, where \(u_i\) is such that: (i) \(u_i = x\), or (ii) \(u_i = \text{parent}(x)\), or (iii) \(e_i = (u_{i-1}, u_i)\) contains \(f\) as an outgoing frond. The first two conditions can clearly be verified in \(O(1)\) time each. Note that in the second case, \(f\) is an outgoing frond of the leg \((p(x) \rightarrow x)\). The third condition holds if \(u_{i-1}\) is an ancestor of \(x\) while \(u_i\) is not. The latter condition can be verified in \(O(1)\) time based on the following well-known fact.

**Lemma 4.1** Let \(v \in V\) and \(nd(v)\) be the number of descendants of \(v\) in the depth-first-search spanning tree. Then \(w\) is a descendant of \(v\) if and only if \(df(s)(v) \leq df(s)(w) < df(s)(v) + nd(v)\).

Using a recursive definition of \(nd(v)\), Procedure DFS can be easily modified to compute \(nd(v), \forall v \in V\), in \(O(|V|)\) time during the depth-first search.

Based on the above discussion, it is easily verified that the time spent on coalescing a millipede is propositional to the number of edges in the millipede. Specifically, we have:

**Lemma 4.2** Let \(\tilde{P} : e_1 \tilde{T}_1 e_2 \tilde{T}_2 \ldots e_h \tilde{T}_h\) be a millipede. Procedure Coalesce takes \(O(h + \sum_{j=1}^{h-1} |E_{\tilde{T}_j}|)\) time to coalesce a section of \(\tilde{P}\), \(e_1 \tilde{T}_1 e_2 \tilde{T}_2 \ldots \tilde{T}_{h-1} e_h\), possibly including a leg in \(\tilde{T}_h\).

**ii. Managing the InFrondLists:**

An edge \(e = (w \leftarrow u)\) is inserted into \(\text{InFrondList}(u)\) when it is encountered as an outgoing frond of vertex \(w\) in Step 1.4 of Procedure DFS. Since \(P_{2h}\) is transformed gradually during the depth-first search, by the time the search backtracks to \(u\), the frond \(e\) may no longer be an outgoing frond of \(w\) but an outgoing frond of a superedge. To determine this superedge efficiently, we proceed as follows.

During the second depth-first search, when \(w\) becomes the current vertex and the frond \(e = (w \leftarrow u)\) is being examined, let \(x\) be the first vertex on \(u \rightarrow w\) such that \(p(x) \neq u\) and \(x\) is not the first child of \(p(x)\). If \(x\) does not exist, then \(e\) is inserted into \(\text{InFrondList}(u)\); otherwise, \(e' = (x \leftarrow u)\) is inserted (Figure 5). In the former case, when the search backtracks to \(u\), the section on the \(u\)-millipede connecting \(u\) with \(w\) or \(u\) with the head \(z\) of the superedge of which \(e\) has become an outgoing frond is to be coalesced, where \(z\) is the first vertex on the \(u\)-millipede that is not a proper ancestor of \(w\). In the latter case, when the search backtracks from \(x\) to \(p(x)\), the frond \(e\) must be a frond in the \(x\)-millipede. Since \(x\) is not the first child of \(p(x)\), the entire millipede is coalesced in Step 1.3, making \(e\) an outgoing frond of the superedge \(e'' = (p(x) \rightarrow x)\). Consequently, when the search backtracks to \(u\), the section on the \(u\)-millipede connecting \(u\) with \(p(x)\) or \(u\) with the head \(z\) of the superedge of which \(e\) has become an outgoing frond will be coalesced, where \(z\) is the first vertex on the \(u\)-millipede that is not a proper ancestor of \(p(x)\). Furthermore, since \(e''\) is the parent edge of \(x\) and it was \(e'\) that was inserted into \(\text{InFrondList}(u)\) in Step 1.4, when \(e'\) is retrieved from \(\text{InFrondList}(u)\) at vertex \(u\), \(p(x)\) and \(e''\) can be retrieved through vertex \(x\) in \(O(1)\) time.

To determine the vertex \(x\) efficiently, Procedure DFS can be elaborated as follows: a number \(\text{path}(v)\) is assigned to every vertex \(v\) during the search so that \(\text{path}(r) = 1; \forall v \in V \setminus \{r\}, \text{path}(v) = \text{path}(p(v))\) if \(v\) is the first child of its parent \(p(v)\), and \(\text{path}(v) = \text{path}(p(v)) + 1\), otherwise.

Let \(w\) be the current vertex of the depth-first search. Using the path values, the tree path \(r \rightarrow w\) can be partitioned into \(h\) sections so that all the vertices on the \(ith\) section, \(1 \leq i \leq h\), have their path values equal to \(i\). Let \(e = (w \leftarrow u)\) be an outgoing frond of \(w\) encountered in Step 1.4 of Procedure DFS. If \(\text{path}(u) = h\) or \(u = p(z)\), where \(z\) is the first vertex with \(\text{path}(z) = h\), then \(e\) is inserted into \(\text{InFrondList}(u)\). Otherwise, the frond inserted is \(f = (x \leftarrow u)\) such that \(x\) is the first vertex with \(\text{path}(x) = \text{path}(u) + 1\) and \(u \neq p(x)\) or \(x\) is the first vertex with \(\text{path}(x) = \text{path}(u) + 2\) and \(u = p(z)\), where \(z\) is the first vertex with \(\text{path}(z) = \text{path}(u) + 1\).

To determine \(x\) in \(O(1)\) time, a stack \(\text{fork}[2..n]\) is maintained so that \(\text{fork}[j]\) contains the first vertex on \(r \rightarrow w\) with \(\text{path}(w) = j, 1 \leq j \leq h = \text{path}(w)\)).

The stack is updated as follows: Initially, \(\text{fork}[1] = r\). Whenever the search advances from a vertex \(w\) to a vertex \(u\) that is not its first child, \(u\) is pushed onto the stack \((u\) is a potential \(x\)). Whenever the search backtracks from a vertex \(u\) to a vertex \(w\) such that \(u\) is not the first child of \(w\), the top element of \(\text{fork}\) is popped. Clearly, \(\text{fork}[j], 1 \leq j \leq h\), contains the vertex, \(v\) that is the first vertex with \(\text{path}(v) = j\). Owing to stack \(\text{fork}\), we have the following lemma.

**Lemma 4.3** Determining and inserting the incoming fronds in \(\text{InFrondList}(u), \forall u \in V\), takes \(O(|E|)\) time.

**Proof:** Determining \(\text{path}(u), \forall u \in V\), takes \(O(|V|)\) time. Maintaining the stack \(\text{fork}\) also takes \(O(|V|)\) time. Determining \(x\) for each frond \(e\) based on the stack takes \(O(1)\) time. Since there are \(|E| - |V| + 1\) fronds, determining and inserting the fronds into \(\text{InFrondList}(u), u \in V\), thus takes \(O(|E|)\) time. 

**iii. Checking if a split operation can be applied:**

\(\forall w \in V, \\text{low}(1)(w), \\text{low}(2)(w),\) and \(\forall x \in V \cup E, \\text{low}(3)(x),\) can clearly be determined in \(O(|E|)\) time.

During the second depth-first search, a new superedge is created when a path on the spine of a mil-

\(\text{Figure 5: Inserting fronds into } \text{InFrondList}(u)\)
lipedc is coalesced. This happens whenever an incoming frond or a child that is not the first child is discovered at the current vertex of the search. Since there are a total of \(|E| - |V| + 1\) incoming fronds and \(|V| - 1\) children, the number of new superedges created is thus at most \(|E|\). The time spent on calculating the low3 value for each of these new superedges is \(O(k)\) where \(k\) is the number of superedges on the path coalesced. Since every edge can be coalesced at most once and there are \(|E|\) superedges originally in \(P_2\) and at most \(|E|\) newly created superedges, the total time spent on calculating the low3 values of all of the newly created superedges is thus \(O(|E|)\).

By Lemmas 3.4, 3.5 and 3.6, checking if a split operation can be applied takes \(O(1)\) time. Since there are at most \(|E|\) split components, there are at most \(|E|\) checks resulting in a split operation being applied and \(2|V|\) checks resulting in no split operation being applied. The total time spent on checking if a split operation can be applied is thus \(O(|V| + |E|) = O(|E|)\).

iv. Determining if a triple bond is to be created:
In Step 1, a triple bond is to be created if \(\exists f = (u_{i+1} \leftrightarrow u_i) \in P_G\). This frond, if exists, must be in \(InFronList(u_0)\). Owing to the structure of \(A[w], w \in V\), and the nature of depth-first search, the fronds are inserted into \(InFronList(u_0)\) in descending order of the \(dfs\) number of their tails. Since there are \(deg_a(u_0)\) incoming fronds of \(u_0\), the total time spent on Step 1 to detect and generate triple bonds is thus \(\sum_{u_0\in V} O(deg_a(u_0)) = O(|E|)\).

In Step 2, a triple bond is to be created if \(\exists f = (u_0 \leftrightarrow lowpt1(u_i))\). Since the outgoing fronds of \(u_0\) are not ordered in \(A[u_0]\), we need an efficient way to determine if \(f\) exists. This is done as follows: a stack \(fstk(u)\) is maintained at each vertex \(u\). When a frond \(f = (w \leftrightarrow u)\) is encountered at \(w\), if the top entry of \(fstk(u)\) is not \(w\), then \(w\) is pushed onto \(fstk(u)\) indicating that a frond \((w \leftrightarrow u)\) has been found. Otherwise, the appearance of \(w\) on \(fstk(u)\) indicates that a frond \((w \leftrightarrow u)\) was found earlier. So, a triple bond \(\{w, u\}\) is created. Similarly, if a virtual edge \((w, u)\) is created for a split component \(w\) and \(u\) appears at the top of \(fstk(u)\), then a triple bond \(\{w, u\}\) is generated. Otherwise, \(w\) is pushed onto \(fstk(u)\). When \(A[u]\) is completely processed, before backtracking to the parent of \(w\), every \(fstk\) on which \(w\) is the top element is popped. Since for each vertex \(u\), there are \(deg_a(u)\) incoming fronds and the tail of each of them is pushed onto and popped out of \(fstk(u)\) at most once, it thus takes \(\sum_{w\in V} O(deg_a(u)) = O(|E|)\) time to manipulate \(fstk(u), \forall u \in V\). Since the total number of edges in the split components is at most \(3|E| - 6\) (Hopcroft and Tarjan (1973)), the number of virtual fronds created in Step 2 is thus \(O(|E|)\). The total time spent on detecting and generating triple bonds in Step 2 is thus \(O(|E|)\).

**Lemma 4.4** Procedure Gen_Splitcomp takes \(O(|E|)\) time to determine all the split-components of \(G\).

**Proof:** Immediate from the above discussion.

**Theorem 4.5** Algorithm Split-components takes \(O(|E|)\) time to generate the split components of \(G\).

**Proof:** Immediate from Lemmas 4.2, 4.3 and 4.4.

5 Conclusion
We have presented a new linear-time algorithm for finding the split components, hence the triconnected components, of an undirected multigraph based on a new graph-transformation technique. The technique could be useful in other context. Moreover, as depth-first search processes the biconnected components in a bottom-up manner, the algorithm can be easily modified so that it would work for graphs that are not biconnected. The algorithm is conceptually simple and makes one less pass over the input graph than the existing best known algorithm of Hopcroft et al. which could mean substantial saving in actual execution time. It is thus of practical interest to implement both algorithms and carry out an empirical study of their performances.

**Acknowledgement**
This research is supported by NSERC under grant NSERC 7811-2009.

**References**


Gutwenger C. & Mutzel, P. (2001), A linear time implementation of SPQR trees, in ‘8th International Symposium on Graph Drawing (GD00)’, Colonial Williamsburg, VA, pp. 77–90.


