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 algorithm 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-***

Copyright ©2012, Australian Computer Society, Inc. This paper appeared at the 18th Computing: Australasian Theory Symposium (CATS 2012), Melbourne, Australia, January-February 2012. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 128, Julian Mestre, Ed. Reproduction for academic, not-for-profit purposes permitted provided this text is included.

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 $\forall v \in V, G - \{v\}$ is a connected graph.

Let $G = (V, E)$ be a biconnected graph and $a, b \in V$. $\forall e, e' \in E$, let $e \sim_{\{a,b\}} e'$ 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 $E_i, 1 \leq i \leq k$, be the equivalence classes w.r.t. $\sim_{\{a,b\}}$. Let $E' = \bigcup_{i=1}^{h} E_i$ and $E'' = \bigcup_{i=h+1}^{k} E_i$ 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 (V_2, respectively) consists of the end-vertices of the edges in E' (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 of 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 $\mathcal{B}_3 \cup \mathcal{T} \cup \mathcal{G}$, where $\mathcal{B}_3$ is a set of triple bonds, $\mathcal{T}$ 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 $\mathcal{B}_3$ 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 $\mathcal{T}$ by merging triangles/polygons that have common edges until no more merging is possible. Then $\mathcal{B} \cup \mathcal{P} \cup \mathcal{G}$ 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 $(u \rightarrow v)$, or a ***frond***, denoted by $(u \hookrightarrow v)$, where u is the ***tail*** and v is the ***head***. The frond $(u \hookrightarrow v)$ 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}. A path in T leading from u to v is denoted by $u \rightsquigarrow v$. When the depth-first search reaches a vertex w, vertex w is called the ***current vertex*** of the search.

A ***millipede***, denoted by $\widehat{T}_0 e_1 \widehat{T}_1 e_2 \widehat{T}_2 \cdots e_k \widehat{T}_k$, is a graph consisting of a path $u_0 e_1 u_1 e_2 u_2 \cdots e_k u_k$ and a set of trees, $\widehat{T}_i, 0 \leq i \leq k$, such that each $\widehat{T}_i$ 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

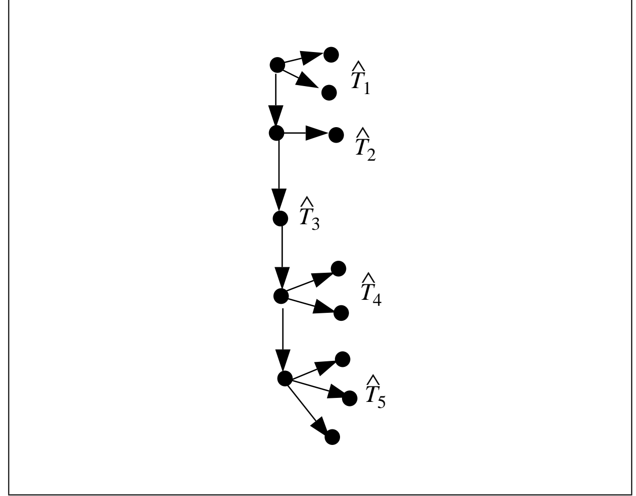

Figure 1: A millipede

$\widehat{T}_0$ is a null tree, the millipede can be denoted by $e_1 \widehat{T}_1 e_2 \widehat{T}_2 \cdots e_k \widehat{T}_k$. 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*** is a graph whose edges are superedges. An ***outgoing frond of a superedge*** e is a frond $(x \hookrightarrow y)$ 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 $\widehat{P}$ are denoted by $\boldsymbol{Out(u)}$, $\boldsymbol{Out(e)}$ and $\boldsymbol{Out(\widehat{P})}$, respectively. An ***edge*** (as oppose to 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, $P_G (= P_G^0)$ is transformed gradually into a sequence of supergraphs $P_G^1, P_G^2, \ldots, R_G^q$ such that every split component in 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 $P_G^j, 1 \leq j \leq q$. However, as the tails of the outgoing fronds of the superedges are not vertices in P_G^j, 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 $e_1 e_2 \ldots e_k$ 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).

$\forall w \in V$,

$low1(w) = \min(\{dfs(w)\} \cup \{dfs(u) | \exists(w \hookrightarrow u)\} \cup \{low1(u) | u \in C(w)\})$;

$low2(w) = \min(\{dfs(w)\} \cup [(\{dfs(u) | \exists(w \hookrightarrow u)\} \cup \{low1(u) | u \in C(w)\} \cup \{low2(u) | u \in C(w)\}) - \{low1(w)\}])$,

where $C(w)$ is the set of children of w.

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 triple bonds 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 $ae_1we_2be'a$, 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 collections 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 pair. Specifically, let $e_1\widehat{T}_1e_2$ 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 $\widetilde{e'}_1$ 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 $\widetilde{e'}_1$ 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 $low1(u)$ and y interchangeably if $dfs(y) = low1(u)$.

First, starting from an arbitrary vertex r, 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, $low1(u)$ and $low2(u)$ are computed. If there is a child v of u such that $low1(v) = low1(u)$, 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

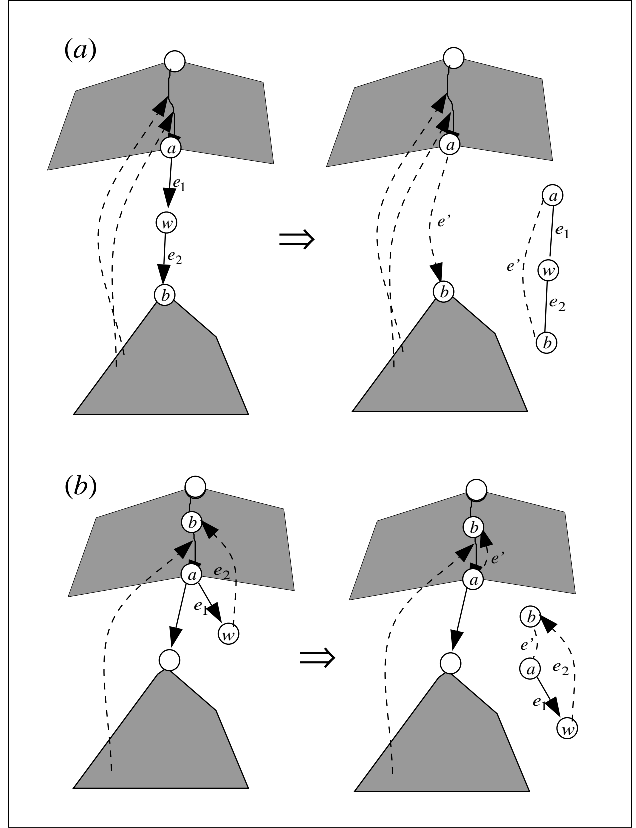

Figure 2:

frond of u, $u \hookrightarrow y$, such that $dfs(y) = low1(u)$ 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 $\langle V_{T_u} \rangle$ 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 $\widehat{P}_u : \widehat{T}_0e_1\widehat{T}_1 \cdots e_k\widehat{T}_k$, 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 $u_0 = u$; u_{i+1} is a first descendant of u_i, $0 \le i < k$, and u_k has an outgoing frond $f = (u_k \hookrightarrow y)$ such that $dfs(y) = low1(u)$. Furthermore, if $k > 0$, then $e_k\widehat{T}_k$ or vertex u_k has an outgoing frond $f' = (x' \hookrightarrow y')$ such that y' is an internal vertex of the path $low1(u) \rightsquigarrow u$.

(ii) for every superedge e in $\widehat{P}_u$, $\widetilde{e}$ 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 $f = (x \hookrightarrow y)$ of $\widehat{P}_u$ (including the terminating vertex u_k), either $dfs(y) < dfs(u)$ or $f = (u_i \hookrightarrow u_{i-1})$, for some $i, 1 \le i \le k$.

Let $\widehat{P} : e_0\widehat{T}_0e_1\widehat{T}_1 \cdots e_k\widehat{T}_k$ be the millipede resulting from concatenating the superedge $e_0 = (w \rightarrow u)$

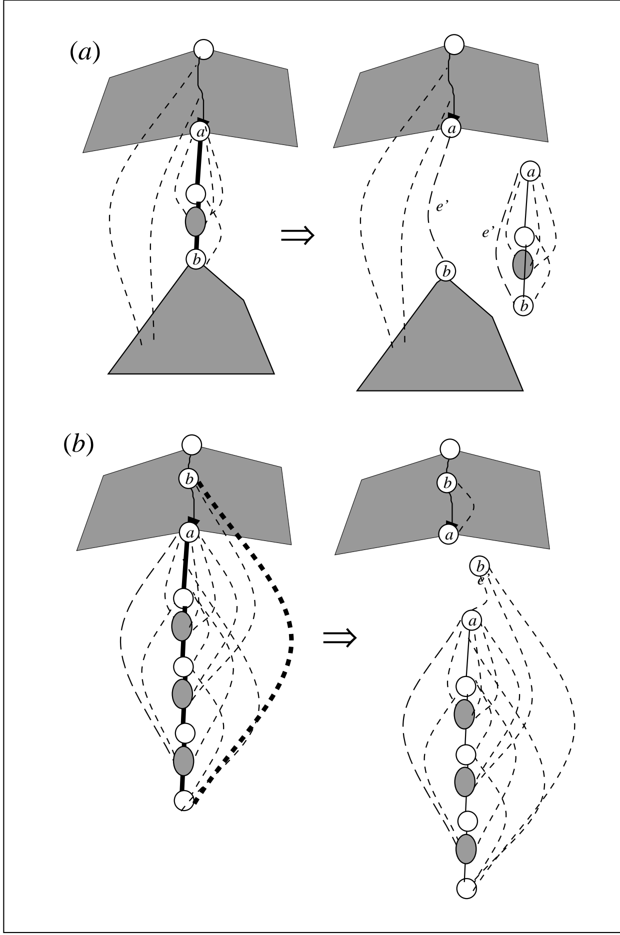

Figure 3:

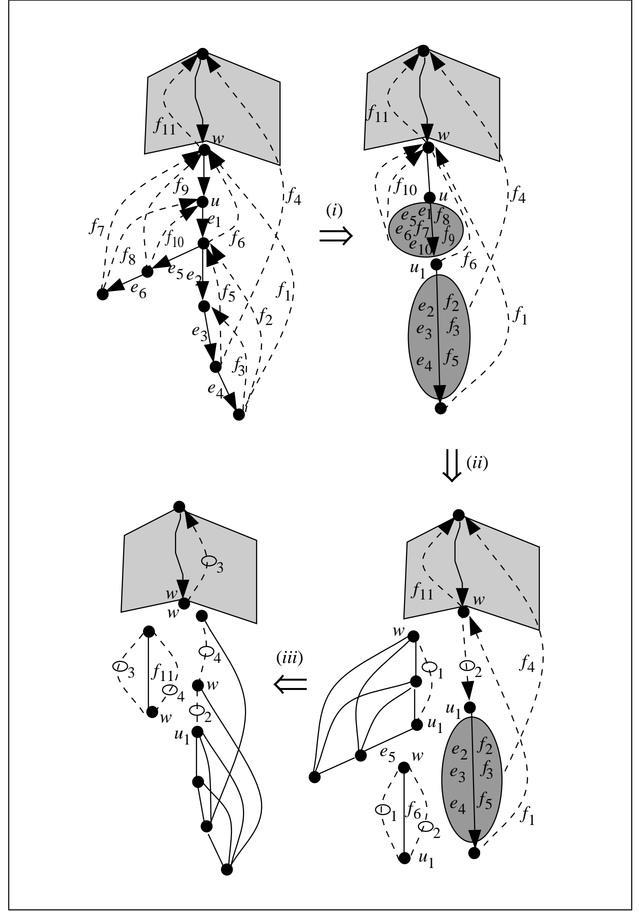

Figure 4: Generating split components from millipede

with $\widehat{P}_u$. If there is no outgoing frond of $e_0\widehat{T}_0e_1$ whose head is outside $e_0\widehat{T}_0e_1$, then $\{w, u_1\}$ is a separation pair and the edges in $e_0\widehat{T}_0e_1$ form a split component (this corresponds to the situation depicted in Figure 3(a)). The section $e_0\widehat{T}_0e_1$ on $\widehat{P}$ is then replaced by a virtual edge $e'_1 = (w, u_1)$. If there is a frond $f' = (u_1 \hookrightarrow w)$, then a triple bond with vertex set $\{w, u_1\}$ is also created (Figure 4(ii)). Let the updated $\widehat{P}$ be $e'_1\widehat{T}_1e_2\widehat{T}_2 \cdots e_k\widehat{T}_k$. If the above condition on outgoing frond applies to the section $e'_1\widehat{T}_1e_2$, then a split component is generated and the section $e'_1\widehat{T}_1e_2$ on $\widehat{P}$ is replaced by a virtual edge. This process is repeated until the aforementioned condition does not hold. Let the resulting millipede be $e'_h\widehat{T}_he_{h+1}\widehat{T}_{h+1} \cdots e_k\widehat{T}_k$. Let $\widehat{P}$ be $e'_h\widehat{T}_he_{h+1}\widehat{T}_{h+1} \cdots e_k\widehat{T}_kf$ such that $f = (u_k \hookrightarrow y)$, where $dfs(y) = low1(u_h)$. If there is no outgoing fronds of $\widehat{P}$ whose head is an internal vertex of the tree-path $low1(u_h) \rightsquigarrow w$ and there is at least one vertex outside $V_{T_{u_h}} \cup \{(w, low1(u_h))\}$, then $\{w, low1(u_h)\}$ is a separation pair and the entire millipede $\widehat{P}$ is removed to produce a split component (Figure 4(iii)) (this corresponds to the situation depicted in Figure 3(b)). A virtual frond $e' = (w \hookrightarrow low1(u_h))$ is introduced to replace $\widehat{P}$. If there is already a frond $f' = (w \hookrightarrow low1(u_h))$ in P_G, then a triple bond with vertex set $\{w, low1(u_h)\}$ is also created. On the other hand, if there is an outgoing fronds of $\widehat{P}$ whose head is an internal vertex of $low1(u_h) \rightsquigarrow w$, then $\widehat{P}$ is coalesced into a superedge $e'_k = (w, u_k)$ if u_h is not a first descendant of w. Otherwise, $\widehat{P}$ becomes the w-millipede $\widehat{P}_w$.

After all the children of w are processed, let $\widehat{P}_w$ be $e'_h\widehat{T}_he_{h+1}\widehat{T}_{h+1} \cdots e_k\widehat{T}_k$. The incoming fronds of w are examined. For each incoming frond $f = (x \hookrightarrow w)$, if $f \in Out(u_j) \cup Out(e_j), h < j \leq k$, then the vertices $u_i, h \leq i < j$, can no longer generate separation pair. The section on $\widehat{P}_w$, $e'_h\widehat{T}_he_{h+1} \cdots \widehat{T}_{j-1}e_j$, is thus coalesced into a superedge $e'_j = (w, u_j)$. Similarly, if f is an outgoing frond of a leg e in some $\widehat{T}_j, h \leq j \leq k$, then the section $e'_h\widehat{T}_he_{h+1} \cdots e_j$ 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_G 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 $dfs(u) > dfs(w)$ and represents an outgoing frond of w, $(w \hookrightarrow u)$, if $dfs(u) < dfs(w)$. 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 $A[w], w \in V$, is much simpler than the *acceptable* 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 $A[w], w \in V$. The split components are determined during this search. The details are presented as Algorithm Split-component below. In the algorithm, $w \oplus \widehat{P}$ denotes the millepede $e_1\widehat{T}_1e_2\widehat{T}_2 \cdots e_k\widehat{T}_k$ if $e_1 = (w \rightarrow u_1)$ and $\widehat{P}$ is the millipede $\widehat{T}_1e_2\widehat{T}_2 \cdots e_k\widehat{T}_k$. 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 $low1(w), low2(w), \forall w \in V$, even though they are calculated during the first search. Moreover, to efficiently determine if a millipede of the form $e_1\widehat{T}_ie_{i+1}$ produces a split component, the following terms are calculated:

$\forall w \in V, low3(w) = \min(\{dfs(w)\} \cup [\{dfs(u) | \exists(w \hookrightarrow u)\} \cup \{low1(u) | u \in C(w) - \{c_1\}\}]$, where c_1 is the first child of w.

$\forall e = (v \rightarrow w), low3(e) = \min(\{dfs(v)\} \cup \{dfs(y) | \exists(x \hookrightarrow y) \in Out(\widetilde{e})\})$.

Specifically, for a vertex w, $low3(w)$ is the highest (closest to the root r) vertex in P_G that is connected to w through a (possibly *null*) tree path that avoids the first child of u following by a frond; for a superedge e, $low3(e)$ is the highest vertex in P_G that is connected to w through an outgoing frond of e.

Algorithm Split-components;
Input: The adjacent lists $A[w]$, $\forall w \in V$.
Output: The split components of $G = (V, E)$.
begin
    count := 1; mark every vertex as *unvisited*;
    DFS($r, \perp$);
end.

Procedure DFS(w, v);
begin /* Initialization */
    mark w as *visited*;
    $dfs(w) := count$; $count := count + 1$;
    $parent(w) := v$; $low2(w) := low3(w) := dfs(w)$;
    $\widehat{P}_w := w$; $Out(w) := \emptyset$; $InFrondList(w) := \emptyset$;

1. **for each** $(u \in A(w))$ **do**
    if $(e = (w \rightarrow u) \in E_T)$ **then** /* a tree edge */
        $\widetilde{e} := \{e\}$; $low3(e) = dfs(w)$;
1.0   DFS(u, w);
        /* Check for split components */
1.1 $\widehat{P}$:= Gen_SplitComp($w \oplus \widehat{P}_u$);
1.2   **if** (u is the first child of w) **then**
            /* $\widehat{P}$ is the w-millipede */
            $\widehat{P}_w := \widehat{P}$; $low1(w) := low1(u)$;
            $low2(w) := \min\{low2(w), low2(u)\}$;
1.3   **else** Coalesce($\widehat{P}, \perp$); /*coalesce the entire $\widehat{P}$*/
            Add $\widehat{P}$ to the $\widehat{T}_i$ subtree rooted at w;
            /* $\widehat{P}$ becomes a leg of $\widehat{T}_i$ */
            if $(low1(u) = low1(w))$ **then**
                $low2(w) := \min\{low2(w), low2(u)\}$
            else $low2(w) := \min\{low2(w), low1(u)\}$;
            $low3(w) := \min\{low3(w), low1(u)\}$
    else if $(e = (w \hookrightarrow u) \in E - E_T)$ **then**
        /* e is an outgoing frond */
        Add e to $Out(w)$;
        /* add e to the list of incoming fronds of u */
1.4   Add e to $InFrondList(u)$;
        if (e is the first frond of w) **then**
            $low1(w) := dfs(u)$
        else if $(dfs(u) > low1(w))$ **then**
            $low2(w) := \min\{low2(w), dfs(u)\}$;
        $low3(w) := \min\{low3(w), dfs(u)\}$;

    else /* e is an incoming frond;
        process it in Step 2 below. */

2. **for each** $((u \hookrightarrow w) \in InFrondList(w))$ **do**
    2.1   Coalesce($\widehat{P}_w, u$);
end;

Procedure Coalesce($\widehat{P}, u$);
begin /* Let $\widehat{P} : e_1\widehat{T}_1e_2\widehat{T}_2 \cdots e_k\widehat{T}_k$. */
 if $(k = 0)$ **then return**; /* $\widehat{P}$ is null */
 if $(u = \perp)$ **then** /* coalesce the entire $\widehat{P}$ */
            $u := u_k$; $all := true$;
    /* coalesce the section of $\widehat{P}$ from w to u */
    $i \leftarrow 1$;
1. **while** $(i < k \wedge u_{i+1} \in Ans(u))$ **do**
            /* coalesce $\widehat{T}_ie_{i+1}$ into e_1 */
        $\widetilde{e}_1 := \widetilde{e}_1 \cup \widetilde{e}_{i+1} \cup (\bigcup_{e \in E_{\widehat{T}_i}} \widetilde{e}) \cup Out(u_i)$;
        $low3(e_1) := \min\{low3(e_1), low3(e_{i+1}), low3(u_i)\}$;
        $i \leftarrow i + 1$;
    endwhile;
2. **if** (u is the tail of an outgoing frond of a leg,
        $e = (u_i, u')$, in $\widehat{T}_i$)
    then /* coalesce the leg e into e_1 */
        $\widetilde{e}_1 := \widetilde{e}_1 \cup \widetilde{e}$;
        $low3(e_1) := \min\{low3(e_1), low3(e)\}$;
    else if (u is the tail of an outgoing frond of
        $e = (u_i, u_{i+1})$)
        then /* coalesce e into e_1 */
            $\widetilde{e}_1 := \widetilde{e}_1 \cup \widetilde{e}$;
            $low3(e_1) := \min\{low3(e_1), low3(e)\}$;
        else if $(all = true)$ **then**
            /* coalesce $\widehat{T}_k$ into e_1 as well */
            $\widetilde{e}_1 := \widetilde{e}_1 \cup (\bigcup_{e \in E_{\widehat{T}_k}} \widetilde{e}) \cup Out(u_k)$;
            $low3(e_1) := \min\{low3(e_1), low3(u_k)\}$;
end; /* of Procedure Coalesce */

Procedure Gen_SplitComp($\widehat{P}$);
begin
    Let $\widehat{P}$ be $e_1\widehat{T}_1e_2\widehat{T}_2 \cdots e_k\widehat{T}_k$, where $e_1 = (u_0, u_1)$.
    $i := 1$;
    if $(k > 1)$ **then**
    /* Check for situation depicted in Fig. 4(ii) */
3.1 **while** $[(i < k) \wedge$
        $\min\{low3(u_i), low3(e_{i+1})\} \geq dfs(u_0)]$ **do**
        output(split component:
            $\{e \mid e$ is an edge of $e_1\widehat{T}_ie_{i+1}\} \cup \{e'_{i+1}\}$),
            where $e'_{i+1} = (u_0, u_{i+1})$ is a new virtual edge;
    1.   **if** $(\exists f = (u_{i+1} \hookrightarrow u_0))$ **then**
            output(triple bond: $\{f, e'_{i+1}, e''_{i+1}\}$),
            where $e''_{i+1} = (u_0, u_{i+1})$ is a new virtual edge;
            replace $e_1\widehat{T}_ie_{i+1}$ in $\widehat{P}$ with e''_{i+1};
            rename e''_{i+1} as e_1;
        else replace $e_1\widehat{T}_ie_{i+1}$ in $\widehat{P}$ with e'_{i+1};
                rename e'_{i+1} as e_1;
        $e_1 := \{e_1\}$; $low3(e_1) := dfs(u_0)$;
        $i := i + 1$;
    endwhile;

    /* Check for situation depicted in Fig. 4(iii) */
3.3 **if** $((low2(u_i) \geq dfs(u_0)) \wedge$
        $((parent(u_0) \neq r) \vee (|C(u_0)| > 1)))$ **then**
        /* the 2nd condition indicates there is a
            vertex outside $\widehat{P}$ */

output(split component:
$\{e \mid e$ is an edge in $\widehat{P}: e_1 T_i e_{i+1} \ldots e_k T_k\} \cup$
$Out(u_k) \cup \{e'_0\}$, where
$e'_0 = (u_0 \hookrightarrow low1(u_i))$ is a new virtual edge;

2. **if** $(\exists f = (u_0, low1(u_i)))$ **then**
output(triple bond: $\{f, e'_0, e''_0\}$), where
$e''_0 = (u_0, low1(u_i))$ is a new virtual edge;
replace $\widehat{P}$ with the null tree $\widehat{T}_0$ with u_0 as the root;
relabel f as e'_0
else replace $\widehat{P}$ with the null tree T_0 with u_0 as the root;
$Out(u_0) := Out(u_0) \cup \{e'_0\}$;
$low3(u_0) := \min\{low3(u_0), low1(u_i)\}$;

return($\widehat{P}$);
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 $\langle V_{T_w} \rangle$ and the fronds that have an end-vertex in T_w has been transformed into a set of isolated millipedes and a w-millipede, $\widehat{P}_w$: $\widehat{T}_0 e_1 \widehat{T}_1 \ldots e_k \widehat{T}_k$, such that:*

(i) *$u_0 = w$; u_{i+1} is a first descendant of $u_i, 0 \le i < k$; $\exists f = (u_k \hookrightarrow y)$ such that $dfs(y) = low1(w)$, and if $k > 0$, then $\exists f' = (x' \hookrightarrow y')$ such that $x' \in Out(e_k \widehat{T}_k) \cup Out(u_k)$ and $low1(w) < dfs(y') < dfs(w)$;*

(ii) *$\forall f = (x \hookrightarrow y) \in Out(\widehat{P}_w) \cup Out(u_k)$, either $dfs(y) < dfs(w)$ or $f = (u_i \hookrightarrow u_{i-1})$, for some $i, 1 \le i \le k$.*

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 $< h (h \ge 1)$.

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 $\langle V_{T_u} \rangle$ 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 $\widehat{P}_u : \widehat{T}_1 e_2 \widehat{T}_2 \ldots e_k \widehat{T}_k$. Then Procedure Gen_SplitComponent is called to process the millipede $w \oplus \widehat{P}_u$ which is $\widehat{P} : e_1 \widehat{T}_1 e_2 \widehat{T}_2 \ldots e_k \widehat{T}_k$, where $e_1 = (u_0, u_1) = (w, u)$.

Within Procedure Gen_SplitComponent, the **while** loop is repeated to produce isolated millipedes until $\widehat{P}$ is reduced to the millipede $e_1 \widehat{T}_h e_{h+1} \ldots e_k \widehat{T}_k$, where either $h = k$ (i.e. $\widehat{P} : e_1 \widehat{T}_k$) or $h < k$ such that $\min\{low3(u_h), low3(e_{h+1})\} < dfs(w)$. If the condition in the **if** statement following the **while** loop holds for $\widehat{P}$, then $\widehat{P}$ 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 $\widehat{P}$ to a superedge $e = (w, u_k)$ (if $\widehat{P}$ has not become the null path) which then becomes a leg of $\widehat{P}_w$ in $\widehat{T}_0$.

On the other hand, if u is the first child of w, then $\widehat{P}$ becomes $\widehat{P}_w$. If $\widehat{P}_w$ has been reduced to a null path, it clearly satisfies the two conditions.

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

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

Since $\widehat{P}_w$ is created from $\widehat{P}_u$, 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 $\widehat{P}_w$. The lemma thus follows. ■

The following lemmas and corollary show that both the split and coalesce operations preserve biconnectivity.

Lemma 3.2 *Let P_G be transformed into a set of isolated millipedes and a supergraph P'_G 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''_G after the split operation is applied. Then P'_G is biconnected implies P''_G 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 $v \oplus \widehat{P}_w$ be $\widehat{T}_0 e_1 \widehat{T}_h e_{h+1} \ldots e_k \widehat{T}_k$ $(1 \le h \le k)$ at that point of time.*

(i) $\nexists$ $f = (x \hookrightarrow y)$ such that $f \in Out(e_1 \widehat{T}_h e_{h+1})$ and $dfs(y) < dfs(v)$ if and only if the millipede $e_1 \widehat{T}_h e_{h+1}$ forms a split component with $\{v, u_{h+1}\}$ as the corresponding separation pair;

(*ii*) *if* $\nexists \quad f = (x \hookrightarrow y)$ *such that* $f \in Out(e_1\widehat{T}_h e_{h+1} \dots e_k\widehat{T}_k f_{k+1})$, *where* $f_{k+1} = (u_k \hookrightarrow z)$ *is the first frond of* u_k, *and* $dfs(z) < dfs(y) < dfs(v)$, *then the millipede* $e_1\widehat{T}_h e_{h+1} \dots e_k\widehat{T}_k f_{k+1}$ *forms a split component with* $\{v, z\}$ *as the corresponding separation pair.*

Proof:
(*i*) Suppose $\nexists \quad f = (x \hookrightarrow y)$ such that $f \in Out(e_1\widehat{T}_h e_{h+1})$ and $dfs(y) < dfs(v)$. Since by Lemma 3.1, $\nexists \quad f = (x \hookrightarrow u_h)$ such that $f \in Out(\widehat{T}_{h+1}e_{h+2}\widehat{T}_{h+2} \dots e_k\widehat{T}_k) \cup Out(u_k)$. The millipede $e_1\widehat{T}_h e_{h+1}$ thus forms a split component with $\{v, u_{h+1}\}$ as the corresponding separation pair.

The converse is obvious.

(*ii*) Clearly, $dfs(z) = low1(u_h)$. Therefore, $\forall f = (x \hookrightarrow y)$ such that $f \in Out(e_1\widehat{T}_h e_{h+1} \dots e_k\widehat{T}_k f_{k+1})$, $dfs(y) \geq dfs(z)$. By assumption, either $dfs(z) \geq dfs(y)$ or $dfs(y) \geq dfs(v)$. Hence, $dfs(y) = dfs(z)$ or $dfs(y) \geq dfs(v)$. The lemma thus follows. ■

Lemma 3.5 *In Statement 3.1 of Procedure Gen_SplitComp,* $\min\{low3(u_i), low3(e_{i+1})\} \geq dfs(u_0)$ *if and only if* $\nexists f = (x \hookrightarrow y) \in Out(e_1\widehat{T}_i e_{i+1})$ *such that* $dfs(y) < dfs(u_0)$.

Proof: Immediate from the definition of $low3(u_i)$ and $low3(e_{i+1})$. ■

Lemma 3.6 *In Statement 3.3 of Procedure Gen_SplitComp,* $((low2(u_i) \geq dfs(u_0)) \wedge ((parent(u_0) \neq r) \vee (|C(u_0)| > 1)))$ *if and only if* $\nexists f = (x \hookrightarrow y) \in Out(e_1\widehat{T}_i e_{i+1} \dots e_k\widehat{T}_k f_{k+1})$, *where* $f_{k+1} = (u_k \hookrightarrow z)$ *is the first frond of* u_k, *such that* $dfs(z) < dfs(y) < dfs(u_0)$ *and there is at least one vertex outside the millipede.*

Proof: The first part follows from the definitions of $low1(u_i)$ and $low2(u_i)$ and Lemma 3.1.

Suppose there is a vertex v outside the millipede $e_1\widehat{T}_i e_{i+1} \dots e_k\widehat{T}_k f_{k+1}$. If $parent(u_0) \neq r$, then v can be a vertex on $r \rightsquigarrow parent(u_0)$. Otherwise, as G, and hence P_G, is biconnected, the root r must have u_0 as the only child. As a result, vertex v is a descendant of u_0 but not of u_i (the child of u_0 on the millipede). It follows that u_0 must has another child implying $|C(u_0)| > 1$. The converse is obvious. ■

In Procedure DFS, after all the split components have been generated using the millipede $\widehat{P} = w \oplus \widehat{P}_u$, if vertex u is not the first child of vertex w or there is a frond $e = (x \hookrightarrow w)$ such that x is a descendant of u, then the remaining $\widehat{P}$ or the tree path $w \rightsquigarrow x$ (possibly modified) in $\widehat{P}$ is coalesced into a new superedge. The following lemma provides the justification.

Lemma 3.7 *Suppose the depth-first search has backtracked from a vertex* w *to its parent vertex* v *and* P_G *has been transformed into a set of isolated millipedes and a supergraph* P'_G. *Let the millipede* $v \oplus \widehat{P}_w$ *in* P'_G *be* $e_1\widehat{T}_h e_{h+1} \dots e_k\widehat{T}_k$ $(1 \leq h \leq k)$ *after the millipede has been processed by Procedure Gen_SplitComp. If* u_h *is not a first descendent of* v *or* $\exists f = (x \hookrightarrow v) \in Out(e_{h+1}\widehat{T}_{h+1} \dots e_k\widehat{T}_k) \cup Out(u_k)$, *then there is no vertex* $z(\neq v)$ *on* $r \rightsquigarrow v$ *such that* $\{u_h, z\}$ *is a separation pair in* P'_G.

Proof: Omitted. ■

In executing Algorithm Split-components, whenever a *split* or *coalesce* operation is performed, the graph is transformed. Let P_G be transformed into a set of isolated millipedes $M_i, 1 \leq i \leq p$, (including the triple bonds) and a supergraph $P'_G = (V', E')$. Suppose P'_G can be further decomposed into a collection of split components $S_i, 1 \leq i \leq q$. Let $\widetilde{M}_i$ ($\widetilde{S}_i$, respectively) denote the set of edges of G that are edges of some superedges in M_i (S_i, respectively), and $\widetilde{P'_G} = \{\widetilde{S}_i \mid 1 \leq i \leq q\}$. The following two lemmas show that the split and coalesce operations preserve the split components of G.

Lemma 3.8 *Let* P_G *be transformed into a set of isolated millipedes and a supergraph* P'_G *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* M *(possibly including a triple bond) and a supergraph* P''_G *after the split operation is applied. 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, $\widetilde{P'_G} = \widetilde{P''_G} \cup \{\widetilde{M}\}$.

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 it is a set of separation pairs of* P'_G. *Moreover, for the collection of split components generated by* S, $\widetilde{P'_G} = \widetilde{P''_G}$.

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.

i. Performing the coalesce operation:

A millipede $\widehat{T}_0 e_1\widehat{T}_1 \dots, e_k\widehat{T}_k$ is represented by a linked list $u_0 - u_1 - \dots - u_k$ (representing its spine) augmented by the following data structures:

•$p(v), \forall v \in V$: the *parent* of v.

•$Out(v), \forall v \in V$: the outgoing fronds of v. Initially, $Out(v) = \emptyset, \forall v \in V$.

•$\widetilde{e_v}, \forall v \in V - \{r\}$, where $e_v = (p(v) \rightarrow v)$. Since the parent edge of v is unique, e_v is determined by v; $\widetilde{e_v}$ 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 $\widetilde{e_v}, v \in V - \{r\}$, so that coalescing two superedges can be done in $O(1)$ time. In particular, in coalescing

a millipede $e_1\widehat{T}_i e_{i+1}$ into a superedge, where $e_1 = (u_0, u_i), e_{i+1} = (u_i, u_{i+1})$; we let $\widetilde{e}_{v_{i+1}} := \widetilde{e}_1 \cup \widetilde{e}_{i+1} \cup (\bigcup_{e \in E_{\widehat{T}_i}} \widetilde{e}) \cup Out(u_i)$, i.e. we let the new superedge be the parent edge of u_{i+1}.

When a frond $f = (x \hookrightarrow u)$ is retrieved from $InFrondList(u)$, 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) $u_i = parent(x)$, 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 $dfs(v) \leq dfs(w) < dfs(v) + nd(v)$.*

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

ii. Managing the InFrondLists:
An edge $e = (w \hookrightarrow u)$ is inserted into $InFrondList(u)$ when it is encountered as an outgoing frond of vertex w in Step 1.4 of **Procedure** DFS. Since P_G 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 $e = (w \hookrightarrow u)$ is being examined, let x be the first vertex on $u \rightsquigarrow w$ 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 $InFrondList(u)$; otherwise, $e' = (x \hookrightarrow u)$ 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 $InFrondList(u)$ in Step 1.4, when e' is retrieved from $InFrondList(u)$ at vertex u, $p(x)$ and $\widetilde{e''}$ can be retrieved through vertex x in $O(1)$ time.

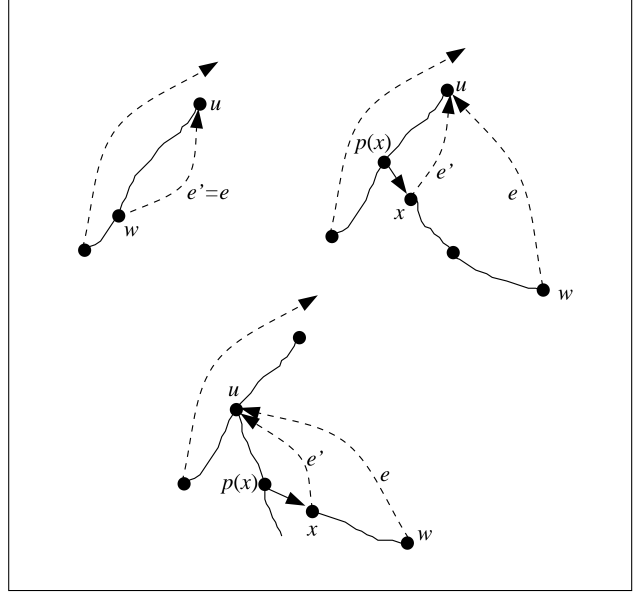

Figure 5: Inserting fronds into $InFrondList(u)$

To determine the vertex x efficiently, **Procedure** DFS can be elaborated as follows: a number $path(v)$ is assigned to every vertex v during the search so that $path(r) = 1$; $\forall v \in V - \{r\}$, $path(v) = path(p(v))$ if v is the first child of of its parent $p(v)$, and $path(v) = path(p(v)) + 1$, otherwise.
Let w be the current vertex of the depth-first search. Using the $path$ values, the tree path $r \rightsquigarrow w$ 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 $e = (w \hookrightarrow u)$ be an outgoing frond of w encountered in Step 1.4 of **Procedure** DFS. If $path(u) = h$ or $u = p(z)$, where z is the first vertex with $path(z) = h$, then e is inserted into $InFrondList(u)$. Otherwise, the frond inserted is $f = (x \hookrightarrow u)$ such that x is the first vertex with $path(x) = path(u) + 1$ and $u \neq p(x)$ or x is the first vertex with $path(x) = path(u) + 2$ and $u = p(z)$, where z is the first vertex with $path(z) = path(u) + 1$.
To determine x in $O(1)$ time, a stack $fork[2..n]$ is maintained so that $fork[j]$ contains the *first* vertex u on $r \rightsquigarrow w$ with $path(u) = j, 1 \leq j \leq h (= path(w))$. The stack is updated as follows: Initially, $fork[1] = r$. 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 $fork$ is popped. Clearly, $fork[j], 1 \leq j \leq h$, contains the vertex, v that is the first vertex with $path(v) = j$. Owing to stack $fork$, we have the following lemma.

Lemma 4.3 *Determining and inserting the incoming fronds in $InFrondList(u)$, $\forall u \in V$, takes $O(|E|)$ time.*
Proof: Determining $path(u), \forall u \in V$, takes $O(|V|)$ time. Maintaining the stack $fork$ 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 $InFrondList(u), u \in V$, thus takes $O(|E|)$ time. ■

iii. Checking if a *split* operation can be applied:
$\forall w \in V, low1(w), low2(w)$, and $\forall x \in V \cup E, low3(x)$, 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-

lipede 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_G 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 $\exists f = (u_{i+1} \hookrightarrow u_0)$ in P_G. This frond, if exists, must be in $InFrondList(u_0)$. Owing to the structure of $A[w], w \in V$, and the nature of depth-first search, the fronds are inserted into $InFrondList(u_0)$ in descending order of the dfs number of their tails. Since there are $deg_{in}(u_0)$ incoming fronds of u_0, the total time spent on Step 1 to detect and generate triple bonds is thus $\sum_{u_0 \in V} O(deg_{in}(u_0)) = O(|E|)$.

In Step 2, a triple bond is to be created if $\exists f = (u_0 \hookrightarrow lowpt1(u_i))$. 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 $f = (w \hookrightarrow u)$ 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 $(w \hookrightarrow u)$ has been found. Otherwise, the appearance of w on $fstk(u)$ indicates that a frond $(w \hookrightarrow u)$ was found earlier. So, a triple bond $\{w, u\}$ is created. Similarly, if a virtual edge (w, u) is created for a split component and w appears at the top of $fstk(u)$, then a triple bond $\{w, u\}$ is generated. Otherwise, w is pushed onto $fstk(u)$. When $A[w]$ 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 $deg_{in}(u)$ incoming fronds and the tail of each of them is pushed onto and popped out of $fstk(u)$ at most once, it thus takes $\sum_{u \in V} O(deg_{in}(u)) = O(|E|)$ time to manipulate $ftsk(u), \forall u \in V$. 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 graph 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

Even, S. (1979), *Graph Algorithms*, Computer Science Press, Potomac, MD.

Fussell, D., Ramachandran, V. and Thurimella, R. (1993), 'Finding triconnected components by local replacement', *SIAM J. Comput.* **22** (3), 587–616.

Gabow, H.N. (2000), 'Path-based depth-first search for strong and biconnected components', *Inf. Process. Lett.* **74** (3-4), 107–114.

Galil, Z. & Italiano, G.F. (1991), 'Reducing edge connectivity to vertex connectivity', *SIGACT News* **2**, 57–61.

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.

Gutwenger C. & Mutzel, P. (2000), 'http://www.ogdf.net/doku.php.'

Hopcroft. J.E. & Tarjan, R.E. (1973), 'Dividing a Graph into Triconnected Components', *SIAM J. Computing* **2**(3), 135–158.

Miller, G.L. & Ramachandran, V. (1992), 'A new graph triconnectivity algorithm and its parallelization', *Combinatorica* **12**, 53–158.

Nagamochi, H. & Ibaraki, T. (1992), 'A Linear Time Algorithm for Computing 3-Edge-Connected Components in a Multigraph', *Japan J. Indust. Appl. Math.* **8**, 163–180.

Saifullah, A. and Ungor, A. (2009), A simple algorithm for triconnectivity of a multigraph, *in* 'CATS'09 (Computing: The Australasian Theory Symposium)', Wellington, New Zealand.

Taoka, S., Watanabe, T. & Onaga, K. (1992), 'A Linear Time Algorithm for Computing all 3-Edge-Connected Components of a Multigraph', *IEICE Trans. Fundamentals* **E75** (3), 410–424.

Tarjan, R.E. (1972), 'Depth-First Search and Linear Graph Algorithms', *SIAM J. Comput.* **1** (2), 146–160.

Tsin, Y.H. (2007), 'A Simple 3-edge-connected Component Algorithm', *Theory of Computing Systems* **40** (2), 125–142.

Tsin, Y.H. (2009), 'Yet another optimal Algorithm for 3-edge-connectivity', *Journal of Discrete Algorithms* **7**(1), 130–146.

Vo, K.P. (1983), 'Finding triconnected components of graphs', *Linear and multilinear algebra* **13**, 119–141.