Theory Comput. Systems **40**, 125–142 (2007)
DOI: 10.1007/s00224-005-1269-4

Theory of Computing Systems

© 2005 Springer Science+Business Media, Inc.

A Simple 3-Edge-Connected Component Algorithm*

Yung H. Tsin

School of Computer Science, University of Windsor,
Windsor, Ontario, Canada N9B 3P4
peter@cs.uwindsor.ca

Abstract. A simple linear-time algorithm for finding all the 3-edge-connected components of an undirected graph is presented. The algorithm performs only one depth-first search over the given graph. Previously best known algorithms perform multiple depth-first searches in multiple phases.

1. Introduction

The notion of k-vertex-connectivity and k-edge-connectivity are introduced to measure the extent to which a graph is connected. For undirected graphs, linear-time algorithms are known only for $k = 2$ or 3 for both vertex- and edge-connectivity. Tarjan presented a simple and elegant linear-time algorithm for 2-vertex-connectivity [8]. His algorithm is based on the powerful graph traversal technique, depth-first search, which was discovered by Hopcroft and himself. The algorithm also solves the 2-edge-connectivity problem in linear time. Gabow revisited depth-first search from a different perspective [2]—the path-based view—and presented new elegant linear-time algorithms for 2-edge-connectivity and 2-vertex-connectivity. For 3-vertex-connectivity, Hopcroft and Tarjan presented a linear-time algorithm [5]. Their algorithm is also based on depth-first search. For 3-edge-connectivity, the first linear-time algorithm was presented by Galil and Italiano [3]. Their method is to reduce 3-edge-connectivity to 3-vertex-connectivity in linear time and then use Hopcroft and Tarjan's linear-time algorithm for 3-vertex-connectivity to solve the problem. Therefore, their algorithm is based on reduction and depth-first search. Two simpler linear-time algorithms were then reported by Taoka et al. [7] and Nagamochi and Ibaraki [6]. Both algorithms also use depth-first search.

* This research was partially supported by NSERC under Grant NSERC-781198.

Taoka et al. [7] compute the 3-edge-connected components in three phases and performs four depth-first searches on the given graph. They divide cut-pairs (a pair of edges whose removal disconnects the given graph) into two types: type-1 and type-2. In phase one all the type-1 cut-pairs are determined. In phase two, the type-2 cut-pairs are determined in three steps. In the first step the given graph is partitioned into disjoint paths so that the edges in every cut-pair lies on the same path. In the second step two important parameters crucial to the detection of type-2 cut-pairs are calculated for every vertex. In the third step the type-2 cut-pairs on each path are determined. In phase three, after adding some new edges to the given graph, the 3-edge-connected components are determined.

Nagamochi and Ibaraki [6] perform a depth-first search on the given graph and then determine for each tree-edge on the depth-first search tree, if the edge and a back-edge form a type-1 cut-pair by counting how many back-edges bypass that tree-edge. After all the type-1 cut pairs are determined, three types of transformations are used to transform the given graph into a smaller graph. The same method is then applied recursively to every non-trivial connected component of the latter. In essence, they find all the type-2 cut-pairs by converting them into type-1 cut-pairs by gradually modifying the given graph. As a result, the total number of edges in all the graphs involved can triple that of the given graph.

Although both algorithms are ingenious, they both require multiple depth-first searches and multiple phases which induce a lot of overhead and thus lack the simplicity and elegancy of the 2-vertex-connectivity algorithms of Tarjan [8] and Gabow [2].

In this paper we present a simple linear-time algorithm which performs only one depth-first search over the given graph. Our algorithm does not distinguish between type-1 and type-2 cut-pairs. Only one type of transformation is used and very little new terminology is introduced. It is conceptually simpler and has the simplicity and elegancy of existing depth-first-search-based 2-vertex-connectivity algorithms.

2. Definitions

Let $G = (V, E)$ be a connected undirected graph where V is its vertex set and E is its edge set. G may contain parallel edges (edges with the same end-vertices) but not self-loops (edges whose two end-vertices are identical). Let $w_1 e_1 w_2 e_2 \cdots w_{k-1} e_{k-1} w_k$ be an alternating sequence of vertices and edges such that $w_i \in V,\ 1 \le i \le k$, and $e_i = (w_i, w_{i+1}) \in E,\ 1 \le i < k$. The sequence is a ***path*** in G if the vertices $w_i,\ 1 \le i \le k$, are distinct; the sequence is a ***cycle*** in G if $w_1 = w_k$ and the vertices $w_i,\ 1 \le i < k$, are distinct, where $k > 1$. When the sequence is a path, vertices w_1 and w_k are the ***terminating vertices***. When $k = 1$, the path is a **null** path. An $x - y$ ***path*** is a path whose terminating vertices are x and y. G is a ***connected*** graph if $\forall u, v \in V$, there exists a $u - v$ path in G. Let $u, v \in V$. Vertices u and v are **3-*edge-connected*** in G, denoted by $u \sim_G v$, if and only if $u = v$ or there exists three edge-disjoint $u - v$ paths in G. It is easily verified that the relation $\sim_G$ is an equivalence relation in V. For $v \in V$, the equivalence class of v with respect to $\sim_G$, denoted by $[v]_{\sim_G}$, is a **3-*edge-connected component*** of G. The ***set of* 3-*edge-connected components*** of G is the set $[V]_{\sim_G} = \{[v]_{\sim_G} \mid v \in V\}$.

A graph $H = (U, F)$ is a ***subgraph*** of a graph $G = (V, E)$ if $U \subseteq V$ and $F \subseteq E$ such that if $e = (u, v) \in F$, then $e = (u, v) \in E$. If $U = V$, then H is a ***spanning subgraph*** of G. A ***connected component*** of a graph G is a maximal connected subgraph of G. Let $U \subseteq V$. The subgraph of G ***induced*** by U, denoted by $\langle U \rangle$, is the maximal subgraph of G whose vertex set is U. A ***tree*** is a connected, undirected graph with no cycle in it.

Let $F \subseteq E$. Then $G - F$ denotes the graph resulting from G after all the edges in F are removed. When $F = \{e\}$, we shall write $G - e$ instead of $G - \{e\}$. A ***bridge*** (or **1-*cut***) in a connected graph $G = (V, E)$ is an edge $e \in E$ such that $G - e$ is a disconnected graph.

Lemma 1 [4, Theorem 3.2]. *Let $G = (V, E)$ be a connected graph and let $e \in E$. Then e is a bridge if and only if e does not lie on any cycle.*

A ***cut-pair*** (or **2-*cut***) in a connected graph $G = (V, E)$ is a pair of edges $e, e' \in E$ such that $G - \{e, e'\}$ is a disconnected graph and neither e nor e' is a bridge in G. A ***cut-edge*** is an edge in a cut-pair. The following lemma is easily verified.

Lemma 2. *Let $G = (V, E)$ be a connected graph and let $e, e' \in E$. If $\{e, e'\}$ is a cut-pair, then e (e', respectively) is a bridge in $G - e'$ ($G - e$, respectively).*

As in [6] and [7], we assume without loss of generality that the graph G is bridgeless throughout this paper.

2.1. *The Absorb-Eject Operation*

The idea underlying our algorithm is to use an operation called *absorb-eject* to transform the given graph G into a *null* graph (an edgeless graph). The null graph is a supergraph with respect to G in the sense that each vertex in the null graph corresponds to a $[v]_{\sim_G}$ of G. Therefore, by keeping track of all the vertices of G merged into each vertex of the null graph, the set of all 3-edge-connected components of G can be determined.

Informally speaking, when an *absorb-eject* operation is performed on an edge $e = (w, u)$ of a graph G' at vertex w, the vertex w absorbs the edge e as well as the vertex u. However, if the degree of u in G', $deg_{G'}(u)$, is 2, vertex w will *spit* out vertex u, making the latter an isolated vertex in the resulting graph. In any case, all the remaining incident edges of vertex u automatically become incident edges of vertex w (self-loops will be omitted). The operation is formally defined as follows.

Definition. Let $G' = (V', E')$ and $e = (w, u) \in E'$ such that either (i) $deg_{G'}(u) = 2$, or (ii) e is not a cut-edge. The graph obtained from G' by applying an ***absorb-eject*** operation on e at w is the graph $G'/e = (V'', E'')$ such that $E'' = E' - E_u \cup E_{w^+}$, where E_u is the set of edges incident upon u in G' and $E_{w^+} = \{f' = (w, z) \mid \exists f \in E_u$, such that $f = (u, z)$ for some $z \in V' - \{w\}\}$, and

$$V'' = \begin{cases} V' & \text{if} \quad deg_{G'}(u) = 2, \\ V' - \{u\} & \text{if } e \text{ is not a cut-edge.} \end{cases}$$

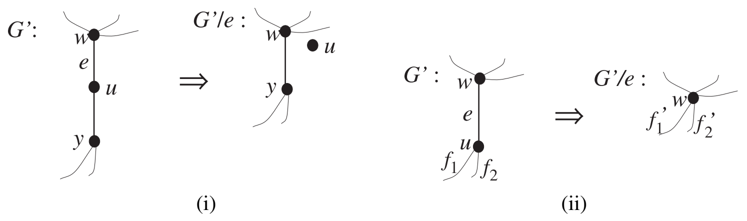

Fig. 1. (i) $deg_{G'}(u) = 2$. (ii) e is not a cut-edge.

The effect of the absorb-eject operation is illustrated in Figure 1. In case (i) vertex u becomes an isolated vertex in G'/e. In case (ii) we say that vertex w ***absorbs*** vertex u. In either case vertex w absorbs edge e. Note that, in case (ii), e is not a cut-edge implies that $deg_{G'}(u) \neq 2$, i.e. cases (i) and (ii) are mutually exclusive. An ***embodiment*** of an edge f is the edge f itself, or the edge $f' = (w, z) \in E_{w^+}$ such that $f = (u, z) \in E_u$, or an embodiment of an embodiment of f. For each $w \in V'$, let $\sigma(w) = \{w\}$ initially, and let $\sigma(w) = \sigma(w) \cup \sigma(u)$ when vertex w absorbs vertex u. Clearly, $\sigma(w)$ denotes the set consisting of vertex w and all the vertices that have been absorbed either by vertex w or by vertices that have been absorbed by vertex w. Let $S \subseteq V'$. $\sigma(S) = \bigcup_{v \in S} \sigma(v)$.

Lemma 3. *Let graph G' be transformed to the graph G'' after a sequence of absorb-eject operations is applied to G'. If G' is bridgeless, then G'' is bridgeless.*

Proof. The absorb-eject operation does not destroy cycles until they become self-loops. The lemma then follows from a simple induction on the number of absorb-eject operations applied to G. □

Lemma 4 [4, Theorem 5.11]. *Let $x, y \in V'$. Then $y \sim_{G'} x$ if and only if there is no cut-pair $\{e', e''\}$ in G' such that vertices x and y belong to different connected components in $G' - \{e', e''\}$.*

Lemma 5. $[V'']_{\sim_{G'/e}} = [V']_{\sim_{G'}} - \{[w]_{\sim_{G'}}\} \cup \{[w]_{\sim_{G'}} - \{u\}\}$, *where* $e = (w, u)$.

Proof. First, if $deg_{G'}(u) = 2$, then by Lemma 4, $\forall x \in V'' - \{u\}, u \not\sim_{G'} x$, which implies that $[u]_{\sim_{G'}} = \{u\}$. On the other hand, $deg_{G'/e}(u) = 0$ implies that $[u]_{\sim_{G'/e}} = \{u\}$. Therefore, $[u]_{\sim_{G'/e}} = [u]_{\sim_{G'}}$. Moreover, as G' and G'/e differ in only the two edges incident upon u in G' being replaced by an edge in G'/e, it is easily verified that, $\forall x \in V'' - \{u\}, [x]_{\sim_{G'/e}} = [x]_{\sim_{G'}}$. Since $w \in V'' - \{u\}$, we thus have $[w]_{\sim_{G'/e}} = [w]_{\sim_{G'}}$ and $u \not\sim_{G'} w$. Therefore, $[w]_{\sim_{G'/e}} = [w]_{\sim_{G'}} - \{u\}$. The lemma thus follows. Next, suppose $e = (w, u)$ is not a cut-edge in G'. $\forall x, y \in V' - \{u\}$, as the absorption of edge e does not result in disconnecting any of the $x - y$ paths or reducing the number of edge-disjoint $x - y$ paths, it is easily verified that $y \sim_{G'} x$ if and only if $y \sim_{G'/e} x$. It follows that $y \in [w]_{\sim_{G'/e}} \Leftrightarrow y \in [w]_{\sim_{G'}}, \forall y \in V''$, and $\forall x \in V" - \{w\}$ such that $[x]_{\sim_{G'/e}} \neq [w]_{\sim_{G'/e}}, y \in [x]_{\sim_{G'/e}} \Leftrightarrow y \in [x]_{\sim_{G'}}, \forall y \in V''$. Since $e = (w, u)$ is not a cut-edge, by Lemma 4, $u \sim_{G'} w$, which implies that $u \in [w]_{\sim_{G'}}$. Moreover, as $u \notin V'', u \notin [w]_{\sim_{G'/e}}$. As a result, $[w]_{\sim_{G'/e}} = [w]_{\sim_{G'}} - \{u\}$. On the other hand, as $u \notin [x]_{\sim_{G'}}$ and $u \notin [x]_{\sim_{G'/e}}$, we have $[x]_{\sim_{G'/e}} = [x]_{\sim_{G'}}$. The lemma thus follows. □

Corollary 5.1. $\{\sigma(S) \mid S \in [V'']_{\sim_{G'/e}}\} = \{\sigma(S) \mid S \in [V']_{\sim_{G'}}\}$.

Proof. Immediate from Lemma 5 and the definition of $\sigma(S)$. □

2.2. *A Brief Description of the Algorithm*

We assume that the reader is familiar with depth-first search [1], [8]. However, for completeness, we give a quick review to some of the related definitions that are used frequently in the subsequent discussion. A depth-first search over a graph $G = (V, E)$ partitions the edge set E into two disjoint subsets T and B. T consists of all those edges of G, called ***tree-edges***, that are traversed by the depth-first search. B consists of the remaining edges, called ***back-edges***. The edge set T induces a rooted spanning tree of G called the ***depth-first spanning tree*** and is denoted by $\boldsymbol{T}_{\mathrm{dfs}}$. A ***subtree*** of T_{dfs} rooted at vertex w, denoted by T_w, is the maximal subgraph of T_{dfs} that is a tree and of which w is the root. The vertex set of T_w is denoted by V_{T_w}. The depth-first search number assigned to vertex w is denoted by $pre(w)$. A back-edge (w, x) is an ***incoming back-edge*** of w if $pre(w) < pre(x)$ and is an ***outgoing back-edge*** of w if $pre(w) > pre(x)$. For each $w \in V$, $lowpt(w) = \min(\{lowpt(u) \mid u \text{ is a child of } w\} \cup \{pre(w') \mid (w, w') \text{ is a back-edge}\} \cup \{pre(w)\})$.

The algorithm starts a depth-first search on G from an arbitrary vertex r. During the search the absorb-eject operation is applied to tree-edges that satisfy conditions (i) or (ii). The purpose of applying the absorb-eject operation to edges satisfying condition (ii) is to bring vertices belonging to the same 3-edge-connected component together under one vertex (stored in the $\sigma(\cdot)$ variable of that vertex). This is because, by Lemma 4, the end-vertices of an edge that is not a cut-edge (condition (ii)) belong to the same 3-edge-connected component. When all the vertices belonging to a 3-edge-connected component have been gathered under a vertex, say u, $deg_{\hat{G}}(u) = 2$, where $\hat{G}$ is the graph to which G has been transformed. The absorb-eject operation is then applied at a vertex adjacent to u, which now satisfies condition (i), to separate u from the rest of the graph. Note that when the absorb-eject operation is performed on a tree-edge $e = (w, u)$ at vertex w, the types of the edges incident upon u are preserved in the sense that if f is a tree-edge (back-edge, respectively) in G', then its embodiment f' is also a tree-edge (back-edge, respectively) in G'/e.

At each vertex w, $lowpt(w)$ is calculated as follows: when the depth-first search enters w the first time, $lowpt(w)$ is initialized to $pre(w)$. Whenever the search backtracks from a child, say u, of w such that $lowpt(u)$ is smaller than the current value of $lowpt(w)$ or encounters a back-edge (w, x) with $pre(x)$ smaller that the current value of $lowpt(w)$, $lowpt(w)$ is updated to $lowpt(u)$ or $pre(x)$. The final value of $lowpt(w)$ is determined when the search backtracks from w to its parent.

It will be shown later that (see Lemma 6) when the depth-first search backtracks from vertex w to its parent, the subgraph of G induced by the vertex set of the subtree rooted at w, i.e. $\langle V_{T_w} \rangle$, has been transformed into a graph consisting of a set of isolated vertices and a tree-path, called the ***w-path***, which is a $w - y$ path such that y is a descendant of w in T_{dfs} and there exists a back-edge $f' = (y, x)$ with $pre(x) = lowpt(w)$ (note that there is no back-edge connecting any two vertices on the w-path). Each isolated vertex corresponds to a 3-edge-connected component of G. Every edge on the w-path has the potential of forming a cut-pair with an edge lying on the $r - w$ tree-path (see

Lemma 7). The w-path is determined as follows: it is initialized to the null path when the depth-first search first enters vertex w. Whenever the depth-first search backtracks from a child vertex, say u (encounters a back-edge, say (w, u), respectively), if $lowpt(u)$ ($pre(u)$, respectively) is smaller than the current value of $lowpt(w)$, then the u-path after being extended to include the tree-edge (w, u) (the null path, respectively) becomes the current w-path. The previous w-path is absorbed by the vertex w with the absorb-eject operation. This is because none of the edges on it can be a cut-edge (see Lemma 8). Therefore, by Lemma 4, all the vertices on it belong to the same 3-edge-connected component as w. Otherwise, the current w-path remains unchanged while the u-path and the tree-edge (w, u) are absorbed by vertex w for a similar reason. In any of the above cases, if $deg(u) = 2$, an absorb-eject operation is applied to the edge (w, u) to make vertex u an isolated vertex. This is because the two edges incident upon u form a cut-pair and node u is thus a 3-edge-connected component such that $\sigma(u)$ is a 3-edge-connected component of G. Since a w-path contains no parallel edges, we denote the path with $P_w : w - w_1 - w_2 - \cdots - w_k$, where w, w_i, $1 \le i \le k$, are the vertices on the path and (w, w_1), (w_i, w_{i+1}), $1 \le i < k$, are the tree-edges connecting them. Figure 2 gives an illustration. In Figure 2(a) a graph G is shown with solid lines representing tree-edges and dotted arrows representing back-edges. In Figure 2(b) the graph to which

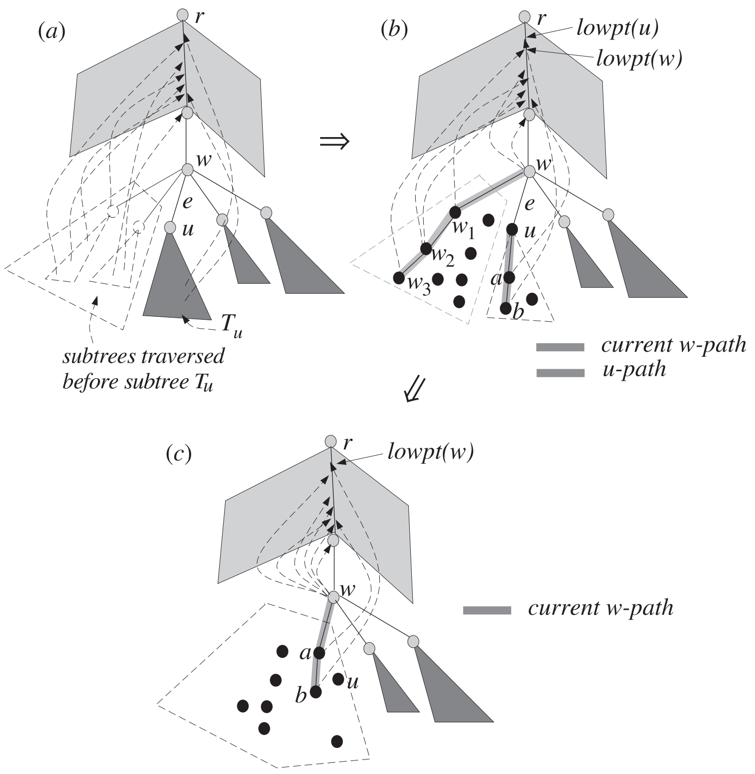

Fig. 2. When depth-first search backtracks from u to w.

G has been transformed when depth-first search backtracks from vertex u to vertex w is shown. Note that the current w-path is $w - w_1 - w_2 - w_3$ and the u-path is $u - a - b$. Since $lowpt(u) < lowpt(w)$, in Figure 2(c) the current w-path is updated to $w - a - b$ while the previous w-path $w - w_1 - w_2 - w_3$ is absorbed by vertex w. Note that an absorb-eject operation was applied to the edge e because $deg(u) = 2$ (Figure 2(b)).

When the depth-first search encounters an incoming back-edge $f' = (w, u)$ that is not a self-loop at vertex w, the back-edge f' must be an embodiment of an incoming back-edge $f = (w, u')$ of w in G. Since u' is a descendant of w in T_{dfs}, u' must reside in a subtree of T_{dfs} rooted at a child of w. The subtree must have been traversed by the depth-first search and hence must have been transformed into a tree-path and a set of isolated vertices such that vertex u' has been absorbed by vertex u. Vertex u cannot be any of the isolated vertices because of the existence of edge f'. It must therefore lie on the tree-path. The tree-path either is the current w-path or must have been absorbed by vertex w earlier. The latter case implies that $u = w$ which means f' is a self-loop, contradicting the assumption. Therefore, vertex u must be a vertex on the current (non-null) w-path. The absorb-eject operation is then applied at w to absorb the section of the current w-path from w to u (Figure 3). This is because no edge lying on this section could be a cut-edge (see Lemma 8). The final w-path is determined when the search backtracks to the parent vertex of w.

When the depth-first search finally backtracks to the root r from the last child, say u, of r, the graph G has been transformed to a graph consisting of a set of isolated vertices and the path $r + P_u$. If the degree of u is 2, an absorb-eject operation would be applied to the tree-edge (r, u). Since $lowpt(r) = 1$, the path $r + P_u$ is then absorbed by r resulting in a set of isolated vertices each corresponding to a 3-edge-connected component of G.

A complete example is given in Figure 4. In Figure 4(i) a graph $G = (V, E)$ and its adjacency list representation are shown. In Figure 4(ii), when the depth-first search backtracks to v_{10} and the back-edge (v_{10}, v_1) is examined, the tree-edge (v_{10}, v_1) is absorbed by v_{10} making $deg(v_{10}) = 2$. As a result, when the search backtracks to v_9, an absorb-eject operation is applied to make v_{10} an isolated vertex (note that

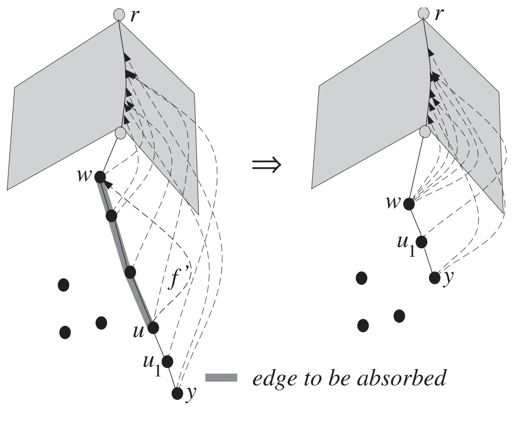

Fig. 3. When an incoming back-edge is examined.

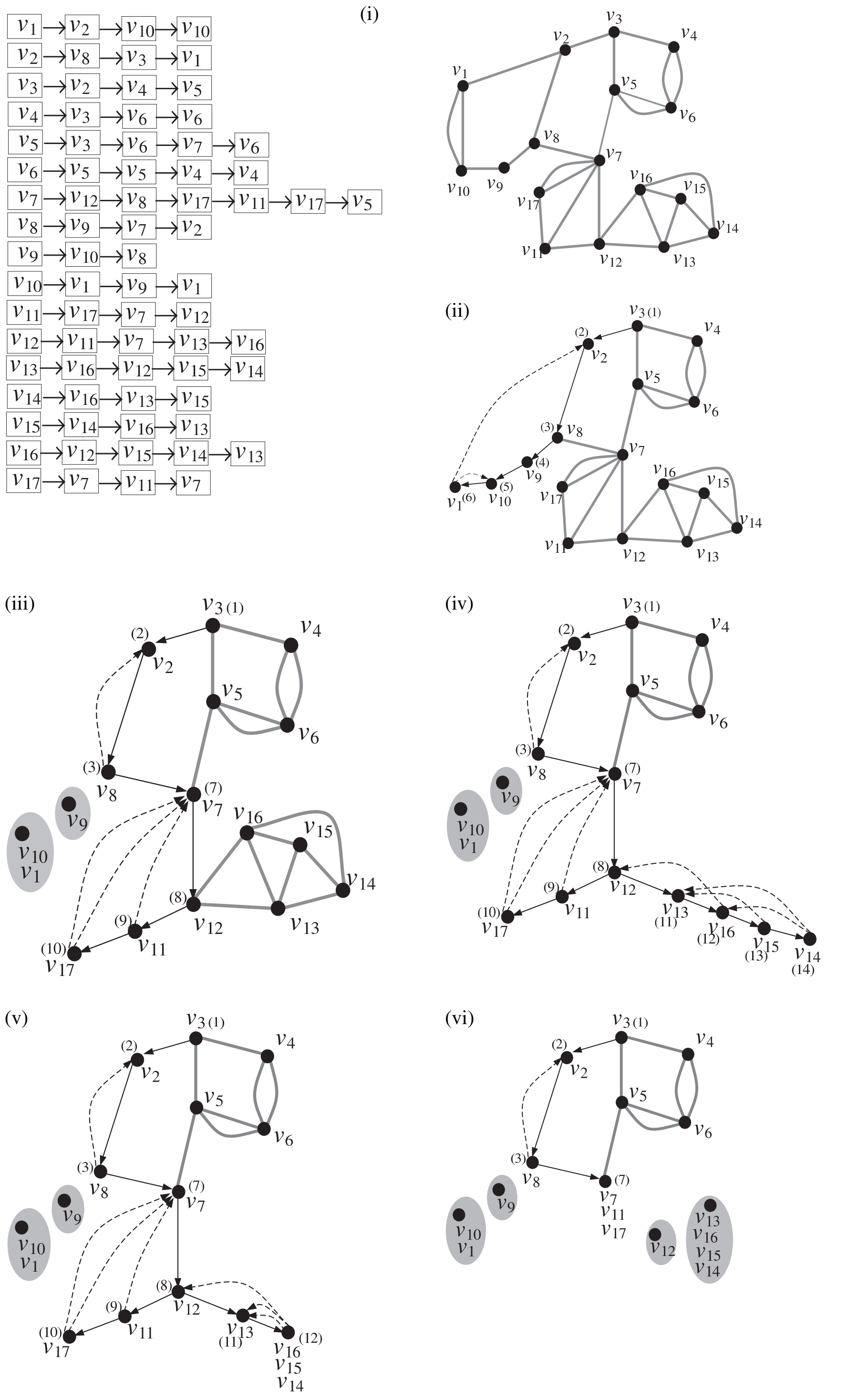
(i)
$v_1 \to v_2 \to v_{10} \to v_{10}$
$v_2 \to v_8 \to v_3 \to v_1$
$v_3 \to v_2 \to v_4 \to v_5$
$v_4 \to v_3 \to v_6 \to v_6$
$v_5 \to v_3 \to v_6 \to v_7 \to v_6$
$v_6 \to v_5 \to v_5 \to v_4 \to v_4$
$v_7 \to v_{12} \to v_8 \to v_{17} \to v_{11} \to v_{17} \to v_5$
$v_8 \to v_9 \to v_7 \to v_2$
$v_9 \to v_{10} \to v_8$
$v_{10} \to v_1 \to v_9 \to v_1$
$v_{11} \to v_{17} \to v_7 \to v_{12}$
$v_{12} \to v_{11} \to v_7 \to v_{13} \to v_{16}$
$v_{13} \to v_{16} \to v_{12} \to v_{15} \to v_{14}$
$v_{14} \to v_{16} \to v_{13} \to v_{15}$
$v_{15} \to v_{14} \to v_{16} \to v_{13}$
$v_{16} \to v_{12} \to v_{15} \to v_{14} \to v_{13}$
$v_{17} \to v_7 \to v_{11} \to v_7$
(ii)
(iii)
(iv)
(v)
(vi)

Fig. 4. An example.

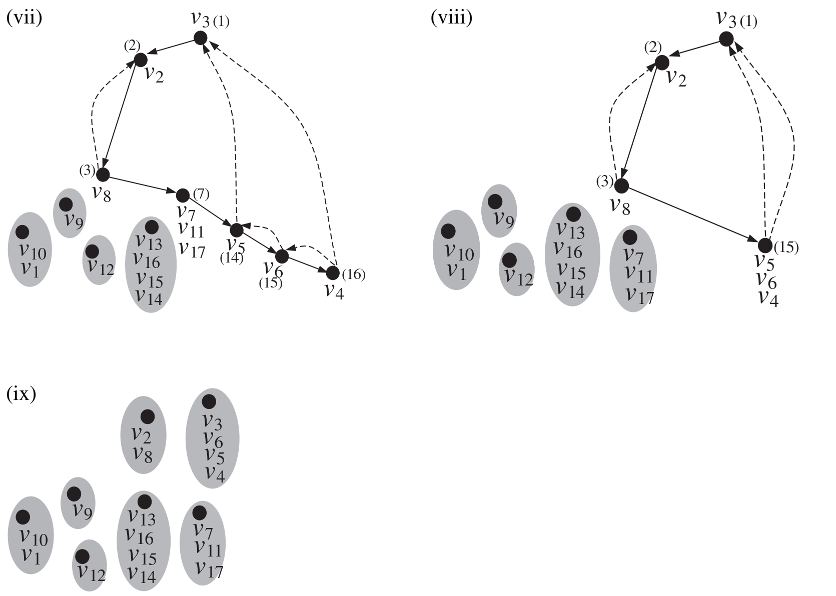

Fig. 4 (*continued*)

$\sigma(v_{10}) = \{v_1, v_{10}\}$). For a similar reason, when the search backtracks to v_8, as $deg(v_9) = 2$, v_9 is made an isolated vertex (see Figure 4(iii)). In Figure 4(iii) the search has just backtracked from v_{11} to v_{12}. The current v_{12}-path is $v_{12} - v_{11} - v_{17}$. In Figure 4(iv), when the back-edge (v_{16}, v_{14}) is examined at v_{16}, the tree-path $v_{16} - v_{15} - v_{14}$ is absorbed by v_{16} resulting in the graph shown in Figure 4(v). In Figure 4(v), owing to the back-edges (v_{13}, v_{16}) (which are the embodiments of back-edges (v_{13}, v_{15}) and (v_{13}, v_{14})), the tree-edge (v_{13}, v_{16}) is absorbed by v_{13} resulting in $deg(v_{13}) = 2$. It follows that an absorb-eject operation is applied at v_{12} to make v_{13} an isolated vertex (note that $\sigma(v_{13}) = \{v_{13}, v_{14}, v_{15}, v_{16}\}$). However, then $deg(v_{12}) = 2$, therefore, when the search backtracks to v_7, an absorb-eject operation is applied to make v_{12} an isolated vertex. Moreover, as $lowpt(v_7) \leq lowpt(v_{12})$, the path $v_7 - v_{11} - v_{17}$ is absorbed by v_7 (see Figure 4(vi)). In Figure 4(vii), when the search backtracks to v_6, owing to the back-edge (v_6, v_4), v_6 absorbs tree-edge (v_6, v_4). Similarly, owing to the back-edge (v_5, v_6), v_5 absorbs tree-edge (v_5, v_6). When the search backtracks to v_8, as $deg(v_7) = 2$, an absorb-eject operation is applied to make v_7 an isolated vertex (see Figure 4(viii), note that $\sigma(v_7) = \{v_7, v_{11}, v_{17}\}$). In Figure 4(viii), when the search backtracks to v_2, owing to the back-edge (v_2, v_8) (which is an embodiment of (v_2, v_1)), v_2 absorbs tree-edge (v_2, v_8) resulting in $deg(v_2) = 2$. Therefore, when the search backtracks to v_3, an absorb-eject operation is applied to make v_2 an isolated vertex (note that $\sigma(v_2) = \{v_2, v_8\}$). Finally, as $lowpt(v_3) \leq lowpt(v_2)$, v_3 absorbs the v_3-path, which consists of the tree-edge (v_3, v_5). The graph $\hat{G} = (\hat{V}, \emptyset)$ is thus produced (Figure 4(ix)).

A formal description of the algorithm is presented below. In the algorithm the following notations are used: Let P_u be the u-path. $P_u - u$ denotes the path resulting from P_u after vertex u is deleted. Let P be a tree-path with v as a terminating vertex and let (w, v) be a tree-edge not lying on P. Then $w + P$ denotes the path resulting from joining P to vertex w with the tree-edge (w, v). Let x, y be any two vertices of P. The section of P from x to y, including x and y, is denoted by $P[x, y]$.

3. The Algorithm

Algorithm 3-edge-connectivity

Input: A bridgeless graph $G = (V, E)$ represented by adjacent lists $L[w]$, $\forall w \in V$.
Output: An edgeless graph $\hat{G} = (\hat{V}, \emptyset)$ such that $\{\sigma(v) \mid v \in \hat{V}\} = [V]_{\sim_G}$.

```
    begin
        count := 1;  3-edge-connect($r$, $\perp$)
    end.
```

Procedure 3-edge-connect(w, v)

```
    begin  mark $w$ as visited;
        $pre(w) := count$;  $count := count + 1$;  $parent(w) := v$;
        $lowpt(w) := pre(w)$; $P_w := w$;     /* initialize $lowpt(w)$ and the $P_w$ path */
    1.  for each ($u$ in $L[w]$) do
            if ($u$ is unvisited) then
                3-edge-connect($u$, $w$);
        1.1     if ($deg(u) = 2$) then
                    $G := G/e$ where $e = (w, u)$ is a tree-edge; $P_u := P_u - u$;
        1.2     if ($lowpt(w) \leq lowpt(u)$) then
        1.3         Absorb-path($w + P_u$);
                else
                    $lowpt(w) := lowpt(u)$;     /* update $lowpt(w)$ */
    1.4             Absorb-path($P_w$);
                    $P_w := w + P_u$     /* update the $P_w$ path */
    1.5.0 else if (($w, u$) is an outgoing back-edge of $w$) then
                    if ($pre(u) < lowpt(w)$) then
    1.5                 Absorb-path($P_w$)
                        $lowpt(w) := pre(u)$;     $P_w := w$;     /* update the $P_w$ path */
    1.6.0       else if (($w, u$) is an incoming back-edge of $w$) then
    1.6                 Absorb-path($P_w[w..u]$);
    end;
```

Procedure Absorb-path(P);

```
    begin       if ($P$ is not the null path) then
                    Let $P$ be $x_0 - x_1 - \cdots - x_k$.     /* $P$ is a tree-path */
                    for $i := 1$ to $k$ do
                        $G := G/e_i$ where $e_i = (x_0, x_i)$
                                                    is an embodiment of edge $(x_{i-1}, x_i)$;
                        $\sigma(x_0) := \sigma(x_0) \cup \sigma(x_i)$
    end;
```

From the description of the algorithm, it is obvious that no absorb-eject operation is applied to any edge on the $r - w$ tree-path before the depth-first search backtracks to the parent of w, for any vertex $w \in V - \{r\}$. Therefore, even though the graph G is transformed gradually during the search, the $r - w$ tree-path in G (i.e. on T_{dfs}) for any vertex w remains intact before the search backtracks to the parent of w. As a result, in subsequent discussion, when we refer to an $r - w$ tree-path, we do not specify to which graph it belongs as long as the search has not backtracked to the parent of w.

Lemma 6. *Let $w \in V - \{r\}$ and let $\hat{G}_w$ be the graph to which G has been transformed when the depth-first search backtracks from vertex w to its parent vertex v. The subgraph of G induced by the vertex set of T_w, i.e. $\langle V_{T_w} \rangle$, has been transformed into a subgraph of $\hat{G}_w$ consisting of a set of isolated vertices and the w-path, $P_w : (w =) w_0 - w_1 - w_2 - \cdot - w_k$, such that*:

(i) *$deg_{\hat{G}_w}(w) \geq 2$ while $deg_{\hat{G}_w}(w_i) \geq 3,\ 1 \leq i \leq k$,*
(ii) *for each back-edge $f = (w_i, x),\ 0 \leq i \leq k$, x lies on the $r - v$ tree-path, and*
(iii) *there exists a back-edge $f = (w_k, z)$ such that $pre(z) = lowpt(w)$.*

Proof. (By induction on the height of w in T_{dfs}.) The base case (i.e. w is of height 0) is obvious.

Suppose the theorem holds true for every vertex with height $< h$, where $h \geq 1$.

Let w be a vertex with height h and $lowpt(w) = pre(z)$. Since G is bridgeless, z is a proper ancestor of w in T_{dfs}, i.e. z lies on the $r - v$ tree-path. Let u' be the vertex through which $lowpt(w)$ is set to $pre(z)$. Then u' is a child of w, or $u' = z$ and there exists a back-edge $f = (w, z)$ in G.

Suppose u' is a child of w. Let u be any child of w. By the induction hypothesis, when depth-first search backtracks from u, the subgraph of G, $\langle V_{T_u} \rangle$, has been transformed into a subgraph of $\hat{G}_u$ consisting of a set of isolated vertices and the u-path $P_u : u - u_1 - u_2 - \cdots - u_k$ such that $deg_{\hat{G}_u}(u) \geq 2$ while $deg_{\hat{G}_u}(u_i) \geq 3$, $1 \leq i \leq k$. Clearly, the isolated vertices will remain isolated in $\hat{G}_w$. if $deg_{\hat{G}_u}(u) = 2$, then vertex u is made an isolated vertex on line 1.1. The vertex will remain isolated in $\hat{G}_w$. Furthermore, if $u \neq u'$, then P_u or $P_u - u$ will eventually be absorbed by vertex w on lines 1.3, 1.4, 1.5 or 1.6. Otherwise, P_u, or $P_u - u$, joins vertex w to form a path $P : w - u'_1 - u'_2 - \cdots - u'_l$. If P is a null path, then conditions (i)—(iii) are clearly satisfied. Otherwise, since $u'_1 - u'_2 - \cdots - u'_l$ is a section of P_u, by the induction hypothesis, for every back-edge $f' = (u'_i, x),\ 1 \leq i \leq l$, x lies on the $r - w$ tree-path. Suppose there exists a back-edge (u'_i, w) for some $1 \leq i \leq l$. Let $e = (u'_p, w)$ be one such back-edge with the largest index p. Then after edge e is encountered and line 1.6 is executed, the path P is reduced to the w-path $P_w : w - u'_{p+1} - \cdots - u'_l$ and all the back-edges (u'_i, w), $1 \leq i \leq p$, are eliminated. Therefore, when the depth-first search backtracks from w to v, for every back-edge $f' = (u'_i, x),\ p < i \leq l$, in $\hat{G}_w$, x lies on the $r - v$ tree-path. Consequently, for every back-edge (w, x) in $\hat{G}_w$, x lies on the $r - v$ tree-path.

Since $deg_{\hat{G}_{u'}}(u'_i) \geq 3,\ p < i \leq l$, we immediately have $deg_{\hat{G}_w}(u'_i) \geq 3$, $p < i \leq l$. If $p \neq l$, then the tree-edges (v, w) and (w, u'_{p+1}) assure that $deg_{\hat{G}_w}(w) \geq 2$. If $p = l$, then by the induction hypothesis and the fact that $lowpt(u') = lowpt(w)$, there exists a back-edge $f'' = (u'_l, z)$. Since w absorbed u'_l, f'' has been transformed to a back-edge $f' = (w, z)$ in $\hat{G}_w$. The tree-edge (v, w) and the back-edge f' assure that $deg_{\hat{G}_w}(w) \geq 2$. Finally, if $p \neq l$ ($p = l$, respectively), then $f'' = (u'_l, z)$ ($f' = (w, z)$, respectively) is the back-edge satisfying condition (iii).

If $u' = z$ and there exists a back-edge $f = (w, z)$ in G, then for every child u of w, the u-path is absorbed by w. $\langle V_{T_w} \rangle$ is thus transformed into a set of isolated vertices and the null path w which clearly satisfies conditions (i)–(iii). □

Lemma 7. *Let $\hat{G}$ denote the graph to which G has been transformed before Procedure Absorb-path is invoked in Instruction* 1.3 (1.4, 1.5 *and* 1.6, *respectively*). *Let $P_x : (w =) \, x_0 - x_1 - \cdots - x_k,\ k \geq 1$, be the actual argument of the procedure at that point of time (i.e. P_x is P_w or $w + P_u$). If $e = (x_i, x_{i+1})$ is a tree-edge lying on P_x such that $\{e, e'\}$ is a cut-pair in $\hat{G}$ for some $e' = (a, b)$, then e' must lie on the $r - w$ tree-path such that (assuming a is the parent of b) (Figure* 5)

(i) *there does not exist a back-edge $f' = (x', y')$ such that x' lies on the $b - x_i$ tree-path and $y' = x_j$ for some $j > i$, and*

(ii) *there does not exist an $s - t$ path in $\hat{G}$ such that s lies on the $r - a$ tree-path while t lies on the $b - x_i$ tree-path and the $s - t$ path and the $r - x_{i+1}$ tree-path share no common vertex, except vertices s and t.*

Proof. Owing to Lemma 6(i) and Instruction 1.1, $deg_{\hat{G}}(x_j) \geq 3$, $1 \leq j \leq k$. As a special case, we have $deg_{\hat{G}}(x_{i+1}) \geq 3$ which implies that there exists a back-edge $f' = (x_{i+1}, v)$ in $\hat{G}$ and hence in $\hat{G}_{u'}$ for some child u' of w. By Lemma 6(ii), v lies

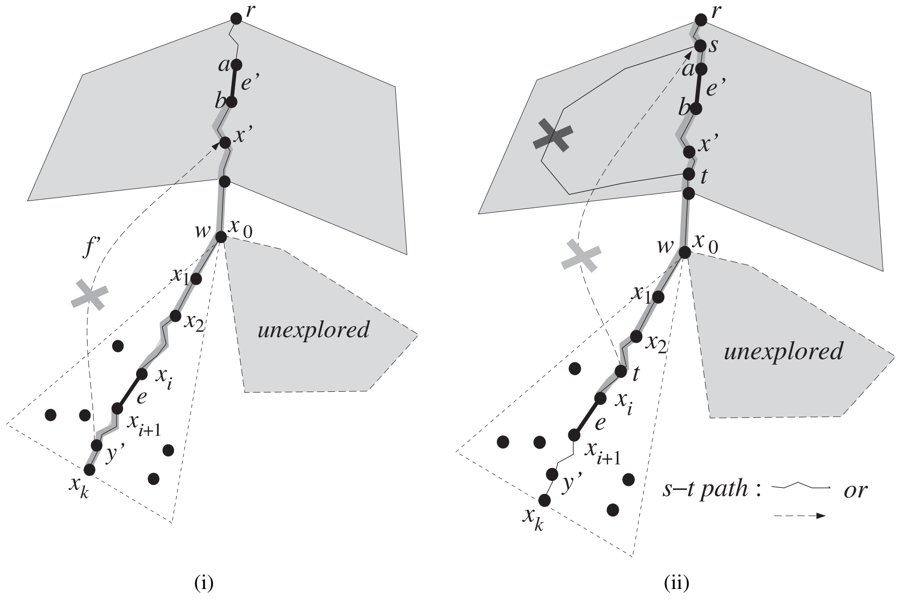

Fig. 5. Lemma 7, conditions (i) and (ii).

on the $r - w$ tree-path. The $v - x_{i+1}$ tree-path and the back-edge f' form a cycle C containing the edge e in $\hat{G}$.

Suppose e' does not lie on the cycle C. Since G is bridgeless, by Lemma 3, $\hat{G}$ is bridgeless. Therefore, $\hat{G} - e'$ is connected and C remains a cycle in $\hat{G} - e'$ containing e. By Lemma 1, e is not a bridge in $\hat{G} - e'$ which contradicts Lemma 2. Therefore, e' lies on cycle C. Suppose $e' = f'$. By Lemma 6(i), $deg_{\hat{G}_{u'}}(x_k) \geq 3$, which implies that $deg_{\hat{G}}(x_k) \geq 3$. Therefore, there exist two back-edges $f_1 = (x_k, v_1)$ and $f_2 = (x_k, v_2)$ in $\hat{G}$. By Lemma 6(ii), both v_1 and v_2 lie on the $r - w$ tree-path. Each f_i and the $v_i - x_k$ tree-path form a cycle containing e. As a result, there is at least one cycle in $\hat{G} - f' = \hat{G} - e'$ containing the edge e (one of the two cycles is eliminated if $i + 1 = k$ and $f' \in \{f_1, f_2\}$). By Lemma 1, e is not a bridge in $\hat{G} - e'$ which contradicts Lemma 2. Therefore, e' must lie on the $v - x_i$ tree-path. Suppose e' lies on the $w - x_i$ tree-path. Then e' lies on P_x. Therefore, $e' = (x_j, x_{j+1})$ for some $j < i$. By applying the preceding argument to e', we have: if $\{e', e''\}$ is a cut-pair, then e'' must lie on a $v' - x_j$ tree-path where v' is a vertex lying on the $r - w$ tree-path. Since, $\{e, e'\}$ is a cut-pair, by letting $e'' = e$ we infer that e lies on the $v' - x_j$ tree-path which implies that $j > i$, a contradiction! Therefore, e lies on the $v - w$ tree-path and hence the $r - w$ tree-path.

Suppose (i) is not true. Then the $x' - y'$ tree-path and the back-edge f' form a cycle containing e but not e'. As a result, the same cycle contains e in $\hat{G} - e'$. By Lemma 1, e is not a bridge in $\hat{G} - e'$ which contradicts Lemma 2.

Suppose (ii) is not true. Then the $s - t$ path and the $s - t$ tree-path form a cycle containing e' but not e in $\hat{G}$. Therefore, the same cycle contains e' in $\hat{G} - e$. By Lemma 1, e' is not a bridge in $\hat{G} - e$ which contradicts Lemma 2. □

Lemma 8. *During the execution of Procedure* 3-*edge-connect*(r), *whenever the absorb-eject operation is applied to an edge* $e = (w, u)$ *at a vertex* w, *either* $deg(u) = 2$ *or* e *is not a cut-edge.*

Proof. First, we remark that $lowpt(w)$ refers to the current value of $lowpt(w)$ throughout this proof.

The absorb-eject operation is applied in Instructions 1.1–1.6.

In Instruction 1.1 the operation is applied to an edge $e = (w, u)$ at vertex w if $deg(u) = 2$.

In Instructions 1.3–1.5 the operation is applied to every edge on a path P_x which is either P_w or $w + P_u$. Let P_x be $w (= x_0) - x_1 - x_2 - \cdots - x_k$, $k \geq 1$. Regardless of whether P_x is P_w or $w + P_u$, by Lemma 6(ii) and (iii), there exists a back-edge $f = (x_k, z')$ such that z' lies on the $r - w$ tree-path. Moreover, there exists a path $P_y : w (= y_0) - y_1 - y_2 - \cdots - y_p$, $p \geq 0$, which is a null path or the path $w + P_u$ if P_x is P_w, or is the path P_w if P_x is $w + P_u$. If P_y is a null path, then there exists a back-edge (w, u) such that $pre(u) < lowpt(w) \leq pre(z')$. Otherwise, by Lemma 6(ii) and (iii), there exists a back-edge (y_p, z) such that z lies on the $r - w$ tree-path and $pre(z) \leq pre(z')$. Now, let $e = (x_i, x_{i+1})$, $0 \leq i < k$, be any edge on P_x. If $w = r$, then the $r - w$ tree-path is null. By Lemma 7, e cannot be a cut-edge. Suppose $w \neq r$. Let $e' = (a, b)$, with $pre(a) < pre(b)$, be any edge on the $r - w$ tree-path. (a) If e' lies on the $z' - w$ tree-path, then the back-edge (w, u) or the path formed by P_y and (y_p, z)

satisfies the condition given in Lemma 7(ii). Therefore, e does not form a cut-pair with e'. (b) If e' lies on the $r - z'$ tree-path, then the back-edge $f = (x_k, z')$ is such that z' lies on the $b - x_i$ path and $k > i$. By Lemma 7(i), e does not form a cut-pair with e'. Hence, e is not a cut-edge.

In Instruction 1.6 the absorb-eject operation is applied to every edge lying on $P_w[w..u]$. Let $e = (x_i, x_{i+1})$ be an edge on $P_w[w..u]$ and let $e' = (a, b)$, with $pre(a) < pre(b)$, be any edge on the $r - w$ tree-path. Then (w, u) is a back-edge satisfying the condition given in Lemma 7(i). Therefore e is not a cut-edge. □

Theorem 1. *When the depth-first search terminates at vertex r, the graph* $G = (V, E)$ *is transformed into a graph* $\hat{G}_r = (\hat{V}, \emptyset)$ *such that* $\{\sigma(v) \mid v \in \hat{V}\} = [V]_{\sim_G}$.

Proof. Since $lowpt(r) = pre(r) = 1$, the u-path of every child u of r is absorbed by r with Instruction 1.1 or 1.3. As a result, when the depth-first search terminates at r, by Lemma 6, the graph G has been transformed into a graph $\hat{G}_r$ consisting of a set of isolated vertices, i.e. $\hat{G}_r = (\hat{V}, \emptyset)$ for some $\hat{V} \subseteq V$.

Let $e_1, e_2, \ldots, e_q$ be the sequence of tree-edges to which the absorb-eject operation is applied to transform G into $\hat{G}_r$. Let $\hat{G}_j = (V_j, E_j)$ be the graph resulting from G after j absorb-eject operations have been performed on G (note that $\hat{G}_0 = G$ and $\hat{G}_q = \hat{G}_r = (\hat{V}, \emptyset)$). By Lemma 8, each e_j, $1 \le j \le q$, either is not a cut-edge or satisfies $deg(u) = 2$ where u is the child vertex among its two end-vertices. By Corollary 5.1, $\{\sigma(S) \mid S \in [V_j]_{\sim_{\hat{G}_j}}\} = \{\sigma(S) \mid S \in [V_{j-1}]_{\sim_{\hat{G}_{j-1}}}\}$, $1 \le j \le q$. By a simple induction on j, we obtain $\{\sigma(S) \mid S \in [V_0]_{\sim_{\hat{G}_0}}\} = \{\sigma(S) \mid S \in [V_q]_{\sim_{\hat{G}_q}}\}$ or equivalently $\{\sigma(S) \mid S \in [V]_{\sim_G}\} = \{\sigma(S) \mid S \in [\hat{V}]_{\sim_{\hat{G}_r}}\} = \{\sigma(v) \mid v \in \hat{V}\}$. As $\sigma(v) = \{v\}, \forall v \in V$ initially, we thus have $[V]_{\sim_G} = \{\sigma(v) \mid v \in \hat{V}\}$. □

It remains to prove that the algorithm runs in linear time.

The graph $G = (V, E)$ is represented by the adjacency list representation in which a linked list $L[w]$ is created for each vertex w. $L[w]$ contains a node for each vertex that is adjacent to w in G.

For each vertex w, in addition to $L[w]$, two pointers, $parent(w)$ and $next(w)$, and two linked lists, $\sigma(w)$ and $\tilde{L}[w]$, are maintained. The $parent(w)$ pointer points at the parent of w in T_{dfs}. The $next(w)$ pointer points at the next vertex on the P_w path or the vertex z such that $pre(z) = lowpt(w)$ if P_w is a null path. The linked list $\sigma(w)$ contains the list of vertices that have been absorbed by w directly or indirectly. The linked list $\tilde{L}[w]$ consists of all the outgoing back-edges of w and of the vertices that have been absorbed by w. Since w is an end-vertex of each of such back-edges, each of the back-edges is represented by a node containing the other end-vertex. $\tilde{L}[w]$ is empty initially. Whenever an outgoing back-edge of w is encountered at w, besides executing Instruction 1.5.0, the back-edge is appended to $\tilde{L}[w]$. Whenever a vertex u is absorbed by w, the entire $\tilde{L}[u]$ is appended to $\tilde{L}[w]$. Although some of the outgoing back-edges of u may now become self-loops at w, we do not examine the list to eliminate them until a later stage when the list is scanned for outgoing back-edges of w to determine if $deg_{\hat{G}_w}(w) = 2$ (details are given in Lemma 11 below).

Since, in the adjacency list representation, every edge $e = (x, y)$ is represented twice, once by the node y in $L[x]$ and once by the node x in $L[y]$, when a vertex u'' absorbs a vertex u' and if $f = (w, u')$ is an outgoing back-edge of u', two modifications must be made to reflect the fact that f has been replaced by its embodiment $f'' = (w, u'')$: the node w in $L[u']$ must be shifted to $L[u'']$ and the node u' in $L[w]$ must be replaced by u''. Appending $\tilde{L}[u']$ to $\tilde{L}[u'']$ takes care of the first modification but not the second (note that $L[u'']$ is practically replaced by $\tilde{L}[u'']$ after the depth-first search backtracks from u" to its parent). It is undesirable to search for node u' in $L[w]$ so as to replace it with node u'' because that would prevent us from getting a linear-time algorithm as the search will have to be performed whenever a back-edge is replaced by an embodiment. This creates the following problem: suppose when the depth-first search backtracks to vertex w in a later stage, the embodiment of f has become $f' = (w, u)$. Since f' is an incoming back-edge of w, it will eventually be encountered at w and Instruction 1.6.0 will be executed. As the node representing f' in $L[w]$ remains u', the incoming back-edge examined in Instruction 1.6.0 is not the desired $f' = (w, u)$ but the original back-edge $f = (w, u')$. Therefore, the algorithm will not work correctly. Fortunately, what we need f' at w for, is to check if a vertex x, hence the tree-edge $(parent(x), x)$, on P_w lies on $P_w[w..u]$. The following lemma shows that we can use f to accomplish the same task.

Lemma 9. *Let $f' = (w, u)$ be an incoming back-edge of w encountered at w during the depth-first search and let $f = (w, u')$ be the incoming back-edge of w in G of which f' is an embodiment. Let x be a vertex on P_w when f' is encountered. Then x lies on $P_w[w..u]$ if and only if x is an ancestor of u' in T_{dfs}.*

Proof. Let $\hat{G}$ be the graph to which G has been transformed when f' is encountered at w. Since the absorb-eject operation is applied to tree-edges only, it is easily verified that for each vertex x in $\hat{G}$, x lies on the $r - u$ tree-path in $\hat{G}$ if and only if x lies on the $r - u$ tree-path in G. Moreover, as the absorb-eject operation is applied at the parent vertex only, x is an ancestor of y in T_{dfs}, for every $y \in \sigma(x)$. Since f' is an embodiment of f, $u' \in \sigma(u)$. Therefore, u is an ancestor of u' in T_{dfs}. Consequently, u lies on the $r - u'$ tree-path in G.

Suppose x lies on $P_w[w..u]$. Then x lies on the $r - u$ tree-path in $\hat{G}$ and hence the $r - u$ tree-path in G. As a result, x is ancestor of u in T_{dfs} and hence an ancestor of u' in T_{dfs}.

Suppose x is an ancestor of u' in T_{dfs}. Then x lies on the $r - u'$ tree-path in G. Since u also lies on the $r - u'$ tree-path, either x lies on the $r - u$ tree-path or u lies on the $r - x$ tree-path. If the latter were true, then u must absorb x in order to absorb u'. This implies that x does not exist on P_w which contradicts the assumption. Therefore, x must lie on the $r - u$ tree-path in G and hence the $r - u$ tree-path in $\hat{G}$. However, both x and u lie on P_w. Hence, x lies on $P_w[w..u]$. □

Therefore, to test if an edge $(parent(x), x)$ on P_w lies on $P_w[w..u]$ in executing Instruction 1.6, we could test if x is an ancestor of u' in T_{dfs} where $f = (w, u') \in B$ of which $f' = (w, u)$ is an embodiment. The latter test can be done in O(1) time providing that $nd(x), \forall x \in V$, is known [9, Corollary 5]. Following [9], Procedure 3-edge-connect(w) can be easily modified to include the computation of $nd(w), \forall w \in V$, in O($|V|$) time.

Lemma 10. *The total time spent on Instruction* 1.6 *to determine all the tree-edges lying on* $P_w[w..u]$ *for all the back-edges* (w, u) *is O*($|E|$).

Proof. Let B_w be the set of incoming back-edges of w. For each $(w, u) \in B_w$, let $l[w..u]$ be the number of edges on $P_w[w..u]$. The edges on $P_w[w..u]$ can be determined by starting from w and following the tree-edges on P_w (using the *next* pointer) until the first edge that is not on $P_w[w..u]$ is encountered. By Lemma 9 and the preceding discussion, this takes O($l[w..u] + 1$) time. Therefore, the total time spent on all the back-edges in $B_w, \forall w \in V$, is $\sum_{w\in V} \sum_{(w,u)\in B_w} \mathrm{O}(l[w..u] + 1)$.

For each $(w, u) \in B_w$, the tree-edges on $P_w[w..u]$ are absorbed by w. Since every tree-edge is absorbed exactly once, every tree-edge belongs to at most one of the $P_w[w..u]$'s. As a result, $\sum_{w\in V} \sum_{(w,u)\in B_w} l[w..u] \le |V| - 1$. Furthermore, as every back-edge in B_w is an embodiment of a distinct incoming back-edge of w in G, $\sum_{w\in V} \sum_{(w,u)\in B_w} 1 \le |B| = |E| - |V| + 1$. Consequently, $\sum_{w\in V} \sum_{(w,u)\in \tilde{B}_w} \mathrm{O}(l[w..u] + 1) = \mathrm{O}(|E|)$. The lemma thus follows. □

Lemma 11. *The logical expression "deg*(u) $= 2$*" in Instruction* 1.1 *takes a total of* O($|E|$) *time to evaluate for all the vertices in* $V - \{r\}$.

Proof. By Lemma 6(iii), all the back-edges incident upon u are outgoing back-edges of u. All these edges including self-loops at u are represented by the linked list $\tilde{L}[u]$ (note that self-loops are not outgoing back-edges). Since $deg(u) = 2$ if and only if the u-path is not null and u has no outgoing back-edge or the u-path is null and u has exactly one outgoing back-edge, it follows that $deg(u) = 2$ if and only if $(pre(next(u)) \ne lowpt(u))$ and $\tilde{L}[u]$ contains only self-loops, or $(pre(next(u)) = lowpt(u))$ and $\tilde{L}[u]$ contains exactly one outgoing back-edge.

Testing if $(pre(next(u)) = lowpt(u))$ takes O(1) time. To test if $\tilde{L}[u]$ contains any back-edge, the list is scanned until either a back-edge is encountered or the list is exhausted. In the latter case, $deg(u) = 2$. In the former case, if $(pre(next(u)) \ne lowpt(u))$, then $deg(u) \ne 2$. Otherwise, the scan is resumed until either another back-edge is encountered or the list is exhausted. In the former case, $deg(u) \ne 2$; in the latter case, $deg(u) = 2$. Since the scanning of $\tilde{L}[u]$ stops at a spot where a decision as to whether $deg(u) = 2$ can be made, there may be some self-loops remaining in $\tilde{L}[u]$. These self-loops will eventually be passed onto an ancestor of u, say x, at which the condition for testing each of these outdated self-loops, say (y, y), becomes $pre(x) < pre(y)$. Therefore, for each y in $\tilde{L}[u]$, (u, y) represents a self-loop or an outdated self-loop if and only if $pre(u) \le pre(y)$. This condition can be checked in O(1) time. The time spent in evaluating $deg(u) = 2$ in Instruction 1.1 is thus at most $b_u + 2$ time units where b_u is the number of self-loops removed.

Since every self-loop is an embodiment of a back-edge and every back-edge in G can become a self-loop at most once, $\sum_{u\in V-\{r\}} b_u \le |B| = |E| - |V| + 1$. Therefore, evaluating $deg(u) = 2$ for all the vertices takes $\sum_{u \in V-\{r\}}(b_u + 2) = \sum_{u \in V-\{r\}} b_u + \sum_{u \in V-\{r\}} 2$ = O($|E| - |V|$)+O($|V|$)=O($|E|$) time. □

Lemma 12. *Procedure Absorb-path*(P) *takes* $O(|P|)$ *time where* $|P|$ *is the number of edges on P.*

Proof. The instruction $G := G/e_i$ can be realized by appending $\tilde{L}[x_i]$ to $\tilde{L}[x_0]$ and the instruction $next(x_0) := next(x_i)$. These operations clearly takes O(1) time. The instruction $\sigma(x_0) := \sigma(x_0) \cup \sigma(x_i)$ also takes O(1) time because $\sigma(x_0) \cup \sigma(x_i)$ can be realized by appending $\sigma(x_i)$ to $\sigma(x_0)$. Therefore, Absorb-path(P) takes O($|P|$) time where $|P|$ is the number of edges on P. □

Theorem 2. *Algorithm* 3-*edge-connectivity takes* $O(|E|)$ *time to determine all the* 3-*edge-connected components of* G.

Proof. In Procedure 3-edge-connect the initialization step before the **for** loop takes O(1) time. Since each vertex is visited exactly once, the total time spent on the initialization step is O($|V|$). The time spent on computing $nd(w)$, $\forall w \in V$, is O($|V|$) (see the remark before Lemma 10).

The **for** loop is executed $deg_G(w)$ times, where $deg_G(w)$ is the degree of w in G. Within the loop, excluding the recursive call, the logical expression "$deg(u) = 2$", the Procedure Absorb-path calls, and the determination of $P_w[w..u]$, it is easily verified that each iteration of the **for** loop takes O(1) time. The total time spent on the **for** loop in all the recursive calls, excluding the aforementioned last three operations, is thus $\sum_{w \in V}$ O($deg_G(w)$) = O($|E|$). By Lemmas 10 and 11, the logical expression "$deg(u) = 2$" and the determination of $P_w[w..u]$ both take O($|E|$) time. Since each tree-edge is absorbed exactly once, by Lemma 12, the total time spent on all the Absorb-path calls is O($|V|$). Therefore, Procedure 3-edge-connect(r) takes O($|E|$) time. Finally, generating the 3-edge-connected components $[V]_{\sim_G}$ can be done in O($|V|$) time by retrieving every vertex v in $\hat{V}$ and outputting the vertices in the $\sigma(v)$ list.

Hence, Algorithm 3-edge-connectivity takes O($|E|$) time to determine all the 3-edge-connected components. □

4. Conclusion

We presented a linear-time algorithm for finding all the 3-edge-connected components of a bridgeless undirected graph. The space complexity is easily verified to be O($|E|$). With slight modification, the cut-pairs which the algorithm uses to produce the 3-edge-connected components can be reported as well. The algorithm can be easily modified to handle graphs containing bridges within the same time and space bound. The algorithm is conceptually simpler than existing algorithms. It is interesting to investigate if a depth-first-search algorithm for 3-edge-connectivity based on the path-based view of Gabow [2] can be deduced from our algorithm.

References

[1] S. Even, *Graph Algorithms*, Computer Science Press, Potomac, MD, 1979.

[2] H. Gabow, Path-based depth-first search for strong and bioconnected components, *Inform. Process. Lett.* **74**, 107–114, 2000 (also appeared as Technical Report CU-CS-890-99, University of Colorado at Boulder, 2000).

[3] Z. Galil and G.F. Italiano, Reducing edge connectivity to vertex connectivity, *SIGACT News* **22**, 57–61, 1991.

[4] F. Harary, *Graph Theory*, Addison-Wesley, Reading, MA, 1969.

[5] J. Hopcroft and R.E. Tarjan, Dividing a graph into triconnected components, *SIAM J. Comput.* **2**, 135–158, 1973.

[6] H. Nagamochi and T. Ibaraki, A linear-time algorithm for computing 3-edge-connected components in a multigraph, *Japan J. Indust. Appl. Math.* **9**, 163–180, 1992.

[7] S. Taoka, T. Watanabe and K. Onaga, A linear-time algorithm for computing all 3-edge-connected components of a multigraph, *IEICE Trans. Fundamentals* **E75**(3), 410–424, 1992.

[8] R.E. Tarjan, Depth-first search and linear graph algorithms, *SIAM J. Comput.* **1**, 146–160, 1972.

[9] Y. H. Tsin and F. Chin, A general program scheme for finding bridges, *Inform. Process. Lett.* **17**, 269–272, 1983.

Received September 27, 2002, *and in final form September* 10, 2004. *Online publication May* 10, 2005.