Journal of Discrete Algorithms 7 (2009) 130–146

Contents lists available at ScienceDirect

Journal of Discrete Algorithms

www.elsevier.com/locate/jda

Yet another optimal algorithm for 3-edge-connectivity

Yung H. Tsin [1]

School of Computer Science, University of Windsor, Windsor, Ontario, Canada

ARTICLE INFO

Article history:
Received 23 June 2007
Received in revised form 8 April 2008
Accepted 28 April 2008
Available online 22 May 2008

Keywords:
Analysis of algorithms
Depth-first search
Graph connectivity
3-edge-connected graph
Edge-connectivity
3-edge-connected component
Cut-pair

ABSTRACT

An optimal algorithm for 3-edge-connectivity is presented. The algorithm performs only one pass over the given graph to determine a set of cut-pairs whose removal leads to the 3-edge-connected components. An additional pass determines all the 3-edge-connected components of the given graph. The algorithm is simple, easy to implement and runs in linear time and space. Experimental results show that it outperforms all the previously known linear-time algorithms for 3-edge-connectivity in determining if a given graph is 3-edge-connected and in determining cut-pairs. Its performance is also among the best in determining the 3-edge-connected components.

1. Introduction

Graph connectivity (edge-connectivity and vertex-connectivity) is a fundamental subject in graph theory that has been extensively studied [1–6,10–13]. A k-edge-connected (k-vertex-connected, respectively) graph is a connected graph which cannot be disconnected by removing less than k edges (vertices, respectively). Graph connectivity has applications in a wide variety of areas such as network reliability, computer vision, and VLSI circuit design.

For 3-edge-connectivity, in addition to the aforementioned applications, it also has applications in physics and quantum chemistry where the Feynman diagram is used [7,8]. A Fyenman diagram consists of a set of vertices and a set of edges. The edges in a Feynman diagram can be divided into two types: V-edges and G-edge. The V-edges are undirected while the G-edges are directed. Every vertex in the diagram is incident with exactly three edges: one V-edge, one G-edge of which the vertex is the tail and one G-edge of which the vertex is the head. A Feynman diagram F is *irreducible* if it cannot be disconnected by the removal of fewer than three G-edges. Let F' be the undirected graph obtained from F by contracting the V-edges and treating the G-edges as undirected edges. Then F is irreducible if and only if F' is 3-edge-connected. In quantum Monte Carlo simulation, it is necessary to determine if a Feynman diagram is irreducible.

In computational biology, 3-edge-connectivity had been used in an FPT (Fixed-Parameter Tractable) algorithm for analyzing protein-protein networks obtained from microarray data [9]. The idea is to determine cut-pairs (a pair of edges whose removal disconnects the given graph) in a graph based on protein-protein network in order to edit clusters and find disjoint cliques that probably represent groups of related proteins. In this context, a cut-pair represents an edge edit (deletion) between two potential cliques.

A number of linear-time algorithms for 3-edge-connectivity had been published [2,6,10,13]. The first one was due to Galil and Italiano [2]. Their method is to reduce 3-edge-connectivity to 3-vertex-connectivity in linear time and then use

E-mail address: peter@uwindsor.ca.

[1] Research partially supported by **NSERC** under grant NSERC-781103.

1570-8667/$ – see front matter
doi:10.1016/j.jda.2008.04.003

Hopcroft and Tarjan' s linear-time depth-first-search-based algorithm for 3-vertex-connectivity [3] to solve the problem. Therefore, their algorithm is based on reduction and depth-first search. Since the 3-vertex-connectivity algorithm is rather complicated, the algorithm of Galil and Italiano, although elegant, is thus quite complicated.

Taoka et al. [10] computes the 3-edge-connected components in three phases and performs four depth-first searches on the given graph. They divide cut-pairs into two types: type-1 and type-2. In phase one, all the type-1 cut-pairs are determined. In phase 2, the type-2 cut-pairs are determined in three steps. In the first step, a depth-first search spanning tree of the given graph is partitioned into disjoint paths so that the edges in each cut-pair lies on the same path. In the second step, for each of the paths, they construct a subgraph of the given graph which consists of the path itself and all the back-paths either originated from the path or terminated at the path (a *back-path* is one consisting of a sequence of edges belonging to the depth-first-search spanning tree, henceforth called *tree-edge*s, following by a single edge that does not belong to the spanning tree, henceforth called *back-edge*). They then transform each such subgraph into a graph consisting of the path and a number of new edges (which they called *in-edge* or *out-edge*) whose end-vertices all lie on the path. 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 et al. [6] performs a depth-first search on the given graph and then determines 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 bypassing 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. However, the total time and space complexity are linear.

Recently, contrary to the multi-pass algorithms of Taoka et al. and Nagamochi et al., Tsin [13] presented a simple algorithm that makes only one pass over the given graph. The algorithm is based on the observation that if two edges have a common vertex of degree 2, then the two edges form a cut-pair. Obviously, not every cut-pair has this property. So a transformation, called *absorb-eject*, is introduced to transform the given graph so that if two edges form a cut-pair and they are not adjacent, then when the depth-first-search backtracks to an end-vertex of the one that is higher in the depth-first-search tree, the graph will eventually be transformed into a graph that has the two edges sharing a common end-vertex of degree 2. Moreover, the common end-vertex is a supervertex in the sense that it consists of all the vertices in a 3-edge-connected component.

In this paper, we present yet another simple linear-time algorithm which performs only one depth-first search over the given graph to determine the set of all cut-edges and a second pass to determine all the 3-edge-connected components. The algorithm does not distinguish between type-1 and type-2 cut-pairs nor does it use any transformation. Experimental results show that our algorithm has better performance than all the other algorithms when the main concern is just to determine if a given graph is 3-edge-connected or to determine the cut-pairs. It outperforms all the others except Tsin [13] when the 3-edge-connected components are to be determined as well. Hence, our algorithm is the choice for applications that call for determining if a given graph is 3-edge-connected or determining all the cut-edges (edges that belong to a cut-pair) of a graph. The applications in physics, chemistry and bioinformatics mentioned earlier are such applications.

2. Basic definitions

We assume the reader has basic knowledge in graph theory. Let $G = (V, E)$ denote an undirected graph in which V is the set of vertices and E is the set of edges. The graph may contain ***parallel edges*** (two or more edges having the same end-vertices) but not self-loops (edges whose end-vertices are identical). Let $u, v \in V$. An ***u-v path*** is a sequence of edges $e_1, e_2, \ldots, e_k$ such that $e_i = (u_{i-1}, u_i)$, $1 \leqslant i \leqslant k$, where $u_i \in V$, $0 \leqslant i \leqslant k$, $u_0 = u$ and $u_k = v$. Two vertices u and v are ***connected*** in G iff there is an $u - v$ path in G. G is ***connected*** iff every two vertices in it are connected. Let $G = (V, E)$ be an undirected connected graph with $|V| \geqslant 2$. An edge is a ***bridge*** in G if its removal results in a disconnected graph. A pair of edges is a ***cut-pair*** in G if their removal results in a disconnected graph and none of them is a bridge. A ***cut-edge*** is an edge in a cut-pair. A ***3-edge-connected component*** of G is a maximal set of vertices of G such that between every two distinct vertices in the set, there are at least three edge-disjoint paths connecting them. Obviously, if two vertices belong to the same 3-edge-connected component, then no removal of bridge or cut-pair in G could result in disconnecting them.

Let $T = (V, E_T)$ be a spanning tree of G created by a depth-first search traversal [11]. T is called a ***DFS-tree*** of G. Let $v \in V$. The ***subtree at v***, denoted by $T(v)$, is the largest subtree of T whose root is v. Let $u, v \in V$. Vertex u is an ***ancestor*** of vertex v iff v is a vertex in the subtree at u. Vertex u is a ***proper ancestor*** of vertex v iff u is an ancestor of v and $u \neq v$. Vertex v is a (***proper***) ***descendant*** of vertex u iff u is an (proper) ancestor of v.

The edges of G lying in T are called the ***tree-edges*** and those lying outside T are called the ***back-edges***. If (u, v) is a back-edge, then either u is an ancestor of v or v is an ancestor of u [11]. Every vertex v is assigned a distinct number, denoted by $dfs(v)$, called its ***depth-first search number*** which is the rank of v in the ordering in which the vertices are visited by the traversal. Let (u, v) be a tree-edge (back-edge, respectively), vertex u is the ***tail*** while vertex v is the ***head*** of the edge if $dfs(u) < dfs(v)$ ($dfs(u) > dfs(v)$, respectively). In the rest of this paper, whenever we denote an edge with (u, v), we shall assume that u is the tail and v is the head. Let (u, v) be a tree-edge, u is the ***parent*** of v and v is a ***child*** of u;

(u, v) is the ***parent edge*** of v and a ***child edge*** of u. An ***incoming back-edge*** of vertex v is a back-edge of which v is the head and an ***outgoing back-edge*** of vertex v is a back-edge of which v is the tail. An x–y ***tree-path*** is the path connecting vertices x and y in T. The ***level*** of a vertex v in T is the number of edges on the r–v tree-path, where r is the root of T.

3. Preliminaries

Let $G = (V, E)$ be an undirected graph and $T = (V, E_T)$ be a DFS-tree of G. For clarity, as with Nagamochi et al. [6], Taoka et al. [10] and Tsin [13], we assume without loss of generality that G is connected and bridgeless throughout this paper.

If e and e' form a cut-pair in G, then at most one of them is a back-edge. This is because removing any two back-edges from G does not result in a disconnected graph owing to the existence of the *DFS*-tree.

Lemma 3.1. *Let $\{e, e'\}$ be a cut-pair in G such that $e = (u, v)$ and $e' = (x, y)$.*

(i) *if e' is a back-edge, then y is an ancestor of u while x is a descendant of v in T;*
(ii) *if both e and e' are tree-edges, then e and e' lie on a tree-path connecting the root to a leaf.*

Proof. A direct consequence of the definition of cut-pair and the properties of DFS-tree. □

Our algorithm is based on the following characterization theorem for cut-pairs.

Theorem 3.2. *Let $e, e' \in E$ be such that $e = (u, v)$, $e' = (x, y)$, $dfs(v) \leqslant dfs(x)$, and e is a tree-edge.*

(i) *if e' is a back-edge, then $\{e, e'\}$ is a cut-pair in G if and only if there does not exist a back-edge $f = (s, t)$ such that $f \neq e'$ and s is a descendant of v while t is an ancestor of u* (*Fig.* 1(a));
(ii) *if e' is a tree-edge, then $\{e, e'\}$ is a cut-pair in G if and only if there does not exist a back-edge (s, t) such that either* (*Fig.* 1(b));
  (a) *s is a descendant of v and not of y while t is an ancestor of u, or*
  (b) *s is a descendant of y while t is a descendant of v and not of y.*

Proof. (i) Suppose e and e' form a cut-pair in G. Then e must be a bridge in $G - e'$ which implies that there does not exist a cycle in $G - e'$ containing e. It follows that there does not exist a back-edge (s, t) in $G - e'$ such that s is a descendant of v while t is an ancestor of u in T. Consequently, there does not exist a back-edge $f = (s, t)$ in G such that $f \neq e'$ and s is a descendant of v while t is an ancestor of u (Fig. 1(a)). Conversely, since $G - e'$ contains no back edges (s, t) such that s is a descendant of v while t is an ancestor of u, there is no cycle in $G - e'$ containing e. As a result, e is a bridge in $G - e'$ which implies that $\{e, e'\}$ is a cut-pair in G.

(ii) See [13]. □

Let E_{cut} be the set of cut-edges in G. To determine all the 3-edge-connected components of G, we shall first determine E_{cut}.

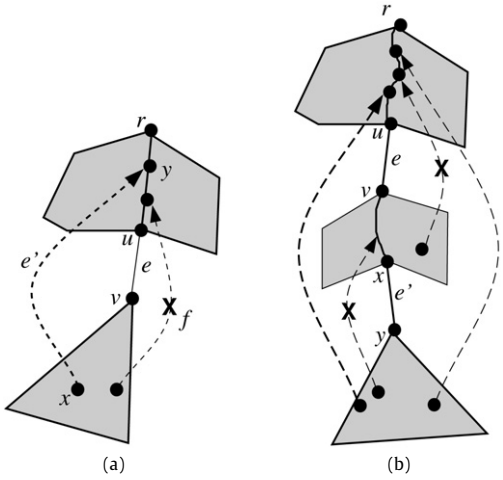


Fig. 1. Illustration of Lemma 3.1 and Theorem 3.2.

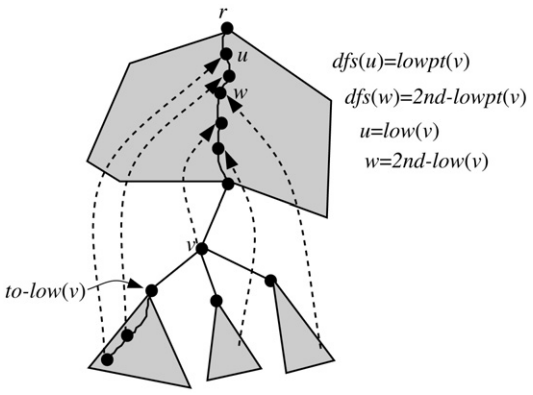


Fig. 2. *lowpt*(v) and *2nd-lowpt*(v).

Definition. Let $e = (x, y)$ be a cut-edge. If either e is a back-edge or e is a tree-edge and there is no tree-edge in $T(y)$ or back-edge having an end-vertex in $T(y)$ that forms a cut-pair with e, then e is called a ***generator***.

Notice that our definition of generator is more general then that of Taoka et al. [10] as we allow back-edges to be generator.

Lemma 3.3. *Let* $\{e, e'\}$ *be a cut-pair such that* $e = (u, v)$, $e' = (x, y)$, $dfs(v) \leqslant dfs(x)$ *and both* e *and* e' *are tree-edges. If* $\{f, e'\}$ *is a cut-pair such that* f *is a generator, then* $\{f, e\}$ *is a cut-pair.*

Proof. Immediate from Theorem 3.2. □

As a direct consequence of Lemma 3.3, every cut-edge belongs to a cut-pair containing a generator. As a result, to determine E_{cut}, it suffices to determine the subset of cut-pairs contain generators.

The following lemma is useful in checking the ancestor/descendant relationship between vertices.

Lemma 3.4. *(See [14].) Let* u, v *be two vertices in a DFS-tree. Vertex* v *is an ancestor of* u *if and only if* $dfs(v) \leqslant dfs(u) \leqslant dfs(v) + nd(v) - 1$, *where* $nd(v)$ *is the number of descendants of* v *in the tree.*

When performing the depth-first search on G, the following two values are calculated at each vertex.

Definition. Let $v \in V$. A v–t ***back-path*** is a path in G consisting of a v–s tree-path, where s is a descendant of v, and a back-edge (s, t) (note that when the v–s tree-path is the null-path, the v–t back-path is the back-edge (v, t)). Then $\boldsymbol{lowpt}(v) = \mathbf{min}(\{dfs(t) \mid \exists a\ v\text{–}t\ back\text{-}path\} \cup \{dfs(v)\})$ [11].

Let ρ_v be a v–t back-path such that $dfs(t) = lowpt(v)$. Then, $\boldsymbol{2nd\text{-}lowpt}(v) = \mathbf{min}(\{dfs(t) \mid (\exists a\ v\text{–}t\ back\text{-}path\, Q)\ Q$ and ρ_v are *edge-disjoint*$\} \cup \{dfs(v)\})$ [13].

Let $\boldsymbol{low}(v)$ ($\boldsymbol{2nd\text{-}low}(v)$, respectively) be the vertex such that $dfs(low(v)) = lowpt(v)$ ($dfs(2nd\text{-}low(v)) = 2nd\text{-}lowpt(v)$, respectively).

An illustration is given in Fig. 2.

Definition. For each $v \in V$, the ***to-low edge*** of v is a back-edge (v, w) such that $dfs(w) = lowpt(v)$ or the first child w of v encountered during the depth-first search such that $lowpt(w) = lowpt(v)$ if there is no such back-edge (v, w). The vertex w is denoted by $\boldsymbol{to\text{-}low}(v)$. In the case where (v, w) is a tree-edge, vertex w is called the ***to-low child*** of v.

Note that if the *to-low edge* is defined by a back-edge and the graph contains parallel edges, then the *to-low edge* is non-unique. In which case, we arbitrary pick one of those (v, w) back-edges satisfying $dfs(w) = lowpt(v)$ as the *to-low* edge.

Definition. The ***to-low path*** of vertex v is the longest path starting from v and consisting of *to-low* edges of descendants of v.

Lemma 3.5. *The to-low path of vertex* v *is a* v-$low(v)$ *back-path.*

Proof. By induction on the level of v in the *DFS*-tree. □

Lemma 3.6. *Let $v \in V - \{r\}$. If $\{e, e'\}$ is a cut-pair such that e' is a tree-edge in $T(v)$ or a non-tree edge having one end-vertex in $T(v)$ while e lies outside $T(v)$. Then e' must lie on the to-low path of v while e lies on the $low(v)$–v tree-path.*

Proof. Lemma 3.1 implies that e must lie on the $r - v$ tree-path while Theorem 3.2(ii)(b) implies that e must lie on the $low(v)$–v tree-path.

Suppose to the contrary that e' does not lie on the *to-low path* of v.

Let $e = (u, w)$ and $e' = (x, y)$. Furthermore, since by Lemma 3.5, the *to-low* path of v is a v–$low(v)$ back-path, let $f = (z', low(v))$ be the back-edge in the *to-low* path of v.

(i) Suppose e' is a back-edge. By Lemma 3.1, e must lie on the y–x tree-path. But e lies outside $T(v)$, therefore it must lie on the y–v tree-path. If x is a vertex on the *to-low* path of v, then as e' is not the *to-low* edge of x, we must have $dfs(y) \geqslant lowpt(x) = lowpt(v)$. On the other hand, if x is not a vertex on the *to-low* path of v, let a be the closest ancestor of x on the *to-low* path of v and b be its child vertex on the a–x tree-path. Since b is not the *to-low* child of a, we must have $dfs(y) \geqslant lowpt(x) \geqslant lowpt(b) \geqslant lowpt(a) = lowpt(v)$. In either case, we have $dfs(y) \geqslant lowpt(v)$. But then the edge f is a back-edge such that $low(v)$ is an ancestor of y and hence an ancestor of u while z' is a descendant of w. This contradicts Theorem 3.2(i).

(ii) Suppose e' is a tree-edge. Let a be the closest ancestor of x on the *to-low* path of v and b be its child vertex on the a–y tree-path. Since b is not the *to-low* child of a, $lowpt(y) \geqslant lowpt(b) \geqslant lowpt(a) = lowpt(v)$ which implies that $low(v)$ is an ancestor of $low(y)$. Let $(x', low(y))$ be the back-edge in the *to-low* path of y. Then by Theorem 3.2(ii)(b), edge e must lie on the $low(y)$–v tree-path. But then the edge f is a back-edge such that $low(v)$ is an ancestor of $low(y)$ and hence of u while z' is a descendant of w and not of y which contradicts Theorem 3.2(ii)(a). □

4. A high-level description of the algorithm

Lemma 4.1. *Let $e = (x, y)$ and $f = (u, v)$ be such that e is a generator and $\{e, f\}$ is a cut-pair. Let $e' = (w, z)$ be another generator lying on the v–x tree-path. Then every edge that forms a cut-pair with e' must lie on the v–w tree-path.*

Proof. We shall consider the case where e is a tree-edge. The case where e is a non-tree edge is similar but simpler.

Since e' is a generator, it does not form a cut-pair with e. By Theorem 3.2(ii), there is a back-edge (s, t) such that either t lies on the z–x tree-path while s is a descendant of y, or t lies on the r–w tree-path while s is a descendant of z but not of y. But $\{e, f\}$ is a cut-pair implies that the former case is impossible owing to Theorem 3.2(ii)(b), and for the latter case t must lie on the v–w tree-path owing to Theorem 3.2(ii)(a). It follows that any edge that forms a cut-pair with e' must lie on the t–w tree-path owing to Theorem 3.2(ii)(b). The lemma thus follows. □

The above lemma shows that the cut-pairs have a nesting structure. Owing to this reason, the stack is a natural data structure to be used in determining cut-pairs. Specifically, during the execution of the algorithm, at each vertex v, a stack, ***stack***(v), consisting of the entries, $[(x_i, y_i), p_i \rightsquigarrow q_i]$, $1 \leqslant i \leqslant k$, is created such that:

(i) $dfs(x_i) \geqslant dfs(y_{i+1})$, $q_i = p_{i+1}$, $1 \leqslant i < k$, $p_1 = low(v)$ and $dfs(q_k) \leqslant dfs(v) \leqslant dfs(x_k)$;
(ii) each (x_i, y_i) is a generator or a potential generator and has the potential of generating cut-pairs with edges on the $p_i \rightsquigarrow q_i$ tree-path;
(iii) every edge outside $T(v)$ that forms a cut-pair with (x_i, y_i) must lie on the $p_i \rightsquigarrow q_i$ tree-path and no edge on the $p_i \rightsquigarrow q_i$ tree-path could form a cut-pair with any edge on the q_i–v tree-path.

For ease of explanation, in the rest of this paper, a back-edge having one end-vertex in $T(v)$, $v \in V$, is considered as an edge lying in $T(v)$.

Definition. Let $v \in V$. Let (x, y) be an edge lying in $T(v)$ and $p \rightsquigarrow q$ be a tree-path lying on $r \rightsquigarrow v$ such that every edge outside $T(v)$ that forms a cut-pair with (x, y) must lie on $p \rightsquigarrow q$ and no edge on $p \rightsquigarrow q$ could form a cut-pair with any edge on the q–v tree-path. Then, $p \rightsquigarrow q$ is called the ***potential path*** of (x, y) at v.

Note that although for every entry $[(x_i, y_i), p_i \rightsquigarrow q_i]$, $1 \leqslant i \leqslant k$, on $stack(v)$, $p_i \rightsquigarrow q_i$, is the potential path of (x_i, y_i) at v, an edge and its potential path at v might not have a corresponding entry on $stack(v)$ if it turns out that no edge on the potential path can form a cut-pair with the edge. Moreover, by Lemma 3.6, (x_i, y_i), $1 \leqslant i \leqslant k$, all lie on the *to-low* path of v while $p_i \rightsquigarrow q_i$, $1 \leqslant i \leqslant k$, all lie on the $low(v)$–v tree-path. Similar stack structures are used in [3,10].

If it turns out that no edge on the $p_i \rightsquigarrow q_i$ tree-path forms a cut-pair with (x_i, y_i), then the entry $[(x_i, y_i), p_i \rightsquigarrow q_i]$ (possibly with $p_i \rightsquigarrow q_i$ being shortened) will be popped out of the stack when the depth-first search backtracks to some ancestor of v on the p_i–v tree-path. Otherwise, let (s, t) be an edge on the $p_i \rightsquigarrow q_i$ tree-path that forms a cut-pair with (x_i, y_i). Then when the depth-first search backtracks to s from t, the top entry on $stack(t)$ must be $[(x_i, y_i), p_i \rightsquigarrow t]$.

The following lemmas provide the basis for updating the stacks.

Lemma 4.2. *Let v be a vertex such that the to-low edge is a back-edge, or the to-low edge is a tree-edge and stack(to-low(v)) is empty. If lowpt(v) < 2nd-lowpt(v), then the tree-path low(v) $\rightsquigarrow$ 2nd-low(v) is the potential path of the to-low edge of v (i.e. the edge (v, to-low(v))) at v.*

Proof. We shall consider the case where the *to-low edge* of v is a back-edge. The case where the *to-low edge* of v is a tree-edge can be proven similarly.

Since the *to-low edge* of v, $(v, low(v))$, is a back-edge, by Lemma 3.1, only edges lying on the tree-path, $low(v) \rightsquigarrow v$, could form cut-pairs with $(v, low(v))$. However, owing to the back-edge $(s, 2nd\text{-}low(v))$, where $s = v$ if v is a leaf or s is a descendant of v, otherwise, only tree-edges lying on the path, $low(v) \rightsquigarrow 2nd\text{-}low(v)$, could form a cut-pair with $(v, low(v))$ by Theorem 3.2(i). Furthermore, no edge on the $low(v) \rightsquigarrow 2nd\text{-}low(v)$ tree-path could form a cut-pair with any edge lying on the $2nd\text{-}low(v) \rightsquigarrow v$ tree-path by Theorem 3.2(ii)(b). The path $low(v) \rightsquigarrow 2nd\text{-}low(v)$ is thus the potential path of $(v, low(v))$ at v. □

Lemma 4.3. *Let v be an internal vertex of the DFS-tree of which the to-low edge is a tree-edge, and $[(x, y), p \rightsquigarrow q]$ be an entry on stack(to-low(v)).*

(i) *If $dfs(q) < 2nd\text{-}lowpt(v)$ and $[(x, y), p \rightsquigarrow q]$ is the top entry on stack(to-low(v)), then the tree-path $q \rightsquigarrow 2nd\text{-}low(v)$ is the potential path of the to-low edge of v at v;*
(ii) *If $dfs(p) < 2nd\text{-}lowpt(v) \leqslant dfs(q)$, then the potential path of (x, y) at v is the tree-path $p \rightsquigarrow 2nd\text{-}low(v)$, and the to-low edge of v is not a generator;*
(iii) *If $2nd\text{-}lowpt(v) \leqslant dfs(p)$, then the potential path of (x, y) at v is the null path, and the to-low edge of v is not a generator.*

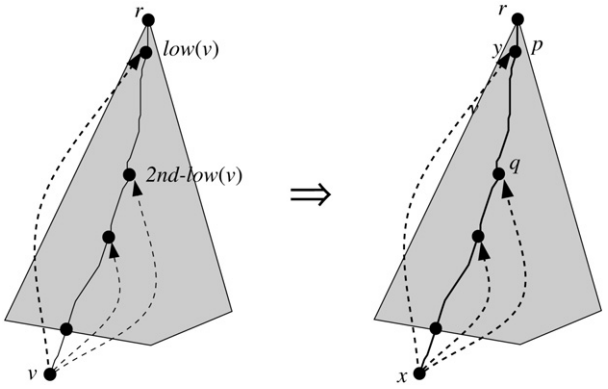
Proof. (i) By Lemma 3.6, any edge lying outside $T(v)$ that forms a cut-pair with an edge on the *to-low path* of v must lie on the $low(v)$–v tree-path.

By the definition of *stack*(to-low(v)), for every entry $[(x', y'), p' \rightsquigarrow q']$ on *stack*(to-low(v)), no edge lying on $q' \rightsquigarrow$ to-low(v) can form a cut-pair with any edge on $p' \rightsquigarrow q'$. Since $[(x, y), p \rightsquigarrow q]$ is the top entry on *stack*(to-low(v)), it follows that no edge lying on $q \rightsquigarrow$ to-low(v) can form a cut-pair with any edge on $low(v) \rightsquigarrow q$. As the *to-low* edge of v lies on $q \rightsquigarrow$ to-low(v), this implies that no edge on $low(v) \rightsquigarrow q$ could form a cut-pair with the *to-low* edge of v. Therefore, any edge that could form a cut-pair with the *to-low* edge of v must lie on $q \rightsquigarrow v$.

Now, as $dfs(q) < 2nd\text{-}lowpt(v)$, the $q \rightsquigarrow 2nd\text{-}low(v)$ tree-path is non-null. Furthermore, either $v = 2nd\text{-}low(v)$ or there exists a back-edge $(s, 2nd\text{-}low(v))$ such that s is a descendant of v and not of to-low(v). In either case, by Theorem 3.2(ii)(a), any edge that forms a cut-pair with the *to-low* edge of v must lie on $q \rightsquigarrow 2nd\text{-}low(v)$, and by Theorem 3.2(ii)(b), no edge on $q \rightsquigarrow 2nd\text{-}low(v)$ could forms cut-pairs with edges on $2nd\text{-}low(v) \rightsquigarrow v$. The tree-path $q \rightsquigarrow 2nd\text{-}low(v)$ is thus the potential path of the *to-low* edge of v at v (An illustration is given in Fig. 3(iii)).

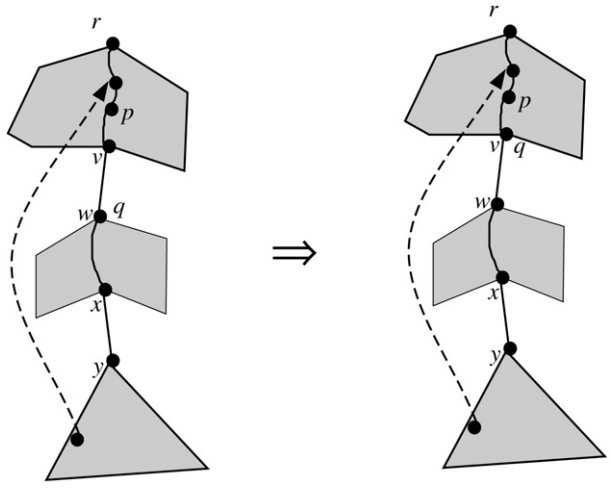
(ii) Since $dfs(q) \leqslant dfs(v)$, therefore, $2nd\text{-}lowpt(v) \leqslant dfs(q)$ implies that $2nd\text{-}lowpt(v) \leqslant dfs(v)$. It follows that either $2nd\text{-}low(v) = v$ or there exists a back-edge $(s, 2nd\text{-}low(v))$ such that s is a descendant of v and not of to-low(v). In the former case, we must have $2nd\text{-}low(v) = q$ which immediately implies that $p \rightsquigarrow 2nd\text{-}low(v)$ is the potential path of (x, y) at v.

In the latter case, as y is a descendant of to-low(v), s is also not a descendant of y. Since $dfs(p) < 2nd\text{-}lowpt(v)$, the tree-path $p \rightsquigarrow 2nd\text{-}low(v)$ is non-null. By Theorem 3.2(ii)(a), owing to the back-edge $(s, 2nd\text{-}low(v))$, every edge that forms

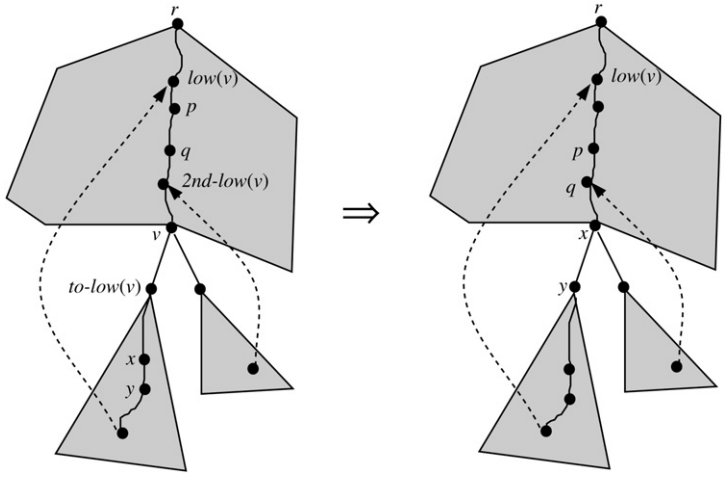


(i)

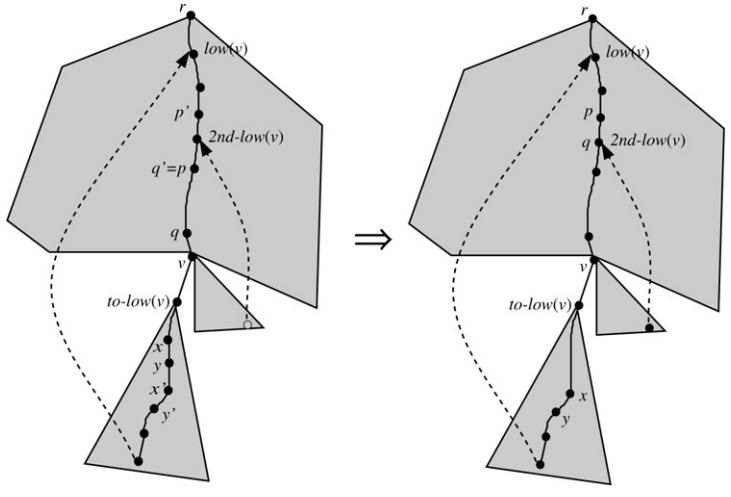
Fig. 3. $[(x, y), p \rightarrow q]$ is the top entry on *stack*(v).



(ii)

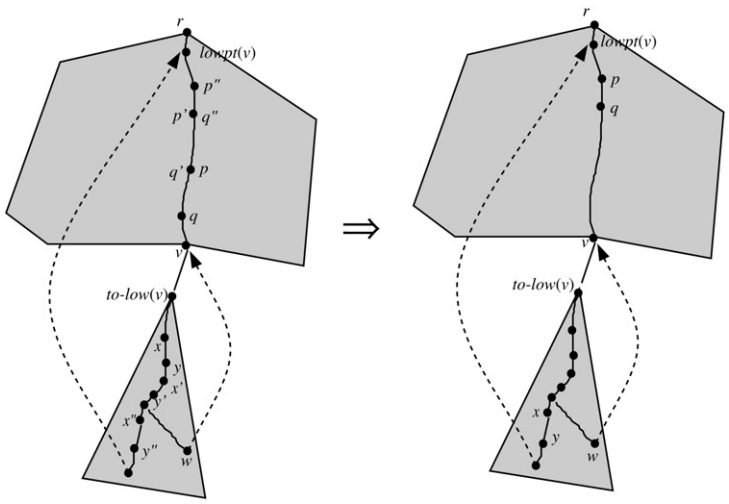


(iii)



(iv)

Fig. 3. (*Continued*)



(v)

Fig. 3. (*Continued*)

a cut-pair with (x, y) must lie on $p \rightsquigarrow 2nd\text{-}low(v)$. Moreover, by Theorem 3.2(ii)(b), no edge lying on $p \rightsquigarrow 2nd\text{-}low(v)$ could form a cut-pair with any edge lying on $2nd\text{-}low(v) \rightsquigarrow v$. The path $p \rightsquigarrow 2nd\text{-}low(v)$ is thus the potential path of (x, y) at v.

Now, as with case (i), any edge that forms a cut-pair with the *to-low edge* of v must lie on $q \rightsquigarrow v$. If $2nd\text{-}low(v) = v$, then $q = v$ which implies that $q \rightsquigarrow v$ is a null-path. Otherwise, owing to the back-edge $(s, 2nd\text{-}low(v))$, no edge on $q \rightsquigarrow v$ could form a cut-pair with the *to-low edge* of v by Theorem 3.2(ii)(a). In either case, the *to-low edge* of v cannot be a generator (an illustration is given in Fig. 3(iv) with $[(x', y'), p' \rightsquigarrow q']$ being the entry of interest).

(iii) Since $dfs(p) < dfs(q) \leqslant dfs(v)$, therefore, $2nd\text{-}lowpt(v) \leqslant dfs(p)$ implies that $2nd\text{-}lowpt(v) < dfs(v)$. It follows that there exists a back-edge $(s, 2nd\text{-}low(v))$ such that s is a descendant of v and not of $to\text{-}low(v)$. But y is a descendant of $to\text{-}low(v)$, therefore s is also not a descendant of y. Let (a, b) be any edge on $p \rightsquigarrow q$. Owing to the back-edge $(s, 2nd\text{-}low(v))$, (a, b) does not form a cut-pair with (x, y) owing to Theorem 3.2(ii)(a). The potential path of (x, y) at v is thus the null path.

Finally, as with the proof of case (ii), the *to-low edge* of v cannot be a generator (an illustration is given in Fig. 3(iv)). □

Lemma 4.4. *Let v be an internal vertex of the DFS-tree such that the to-low edge is a tree-edge, and $[(x, y), p \rightsquigarrow q]$ is an entry on $stack(to\text{-}low(v))$. If (x, y) is a tree-edge and there is an incoming back-edge of v, (u, v), such that u is a descendant of y, then the potential path of (x, y) at v is the null path.*

Proof. Immediate from Theorem 3.2(ii)(b). □

For clarity, we shall first give a high-level description of our algorithm.

The depth-first search traversal begins from an arbitrary vertex r. In general, at each vertex v:

(i) if v is a leaf in T, then the *to-low edge* of v is a back-edge. If $lowpt(v) < 2nd\text{-}lowpt(v)$, by Lemma 4.2, the back-edge $(v, low(v))$ is a potential generator with $low(v) \rightsquigarrow 2nd\text{-}low(v)$ being its potential path. As a result, the entry $[(v, low(v)), low(v) \rightsquigarrow 2nd\text{-}low(v)]$ is pushed onto $stack(v)$ (Fig. 3(i));

(ii) if v is not a leaf in T, then whenever the traversal returns from a child w of v, all cut-pairs lying within $T(w)$ of which one cut-edge is a generator are determined. Moreover, $stack(w)$ is also created. If the top entry $[(x, y), p \rightsquigarrow q]$ on $stack(w)$ satisfies $q = w$ (Fig. 3(ii)), then this confirms that the edges (v, w) and (x, y) form a cut-pair and the two edges are marked as cut-edges accordingly. Furthermore, the top entry is updated to $[(x, y), p \rightsquigarrow v]$ if $p \rightsquigarrow v$ is not the null path; it is deleted otherwise.

When the depth-first traversal has gone through the sub-trees of all the children of v, if v has no *to-low* child (i.e. the *to-low* edge of u is a back-edge), then the *to-low* path of v consists of the *to-low* edge only. By Lemma 3.6, no edge in $T(v)$ (a back-edge with one end-vertex in $T(v)$ is considered as lying in $T(v)$) other than the *to-low* edge of v could form a cut-pair with an edge outside $T(v)$. The initial $stack(v)$ is thus empty. If $low(v) < 2nd\text{-}lowpt(v)$, then by Lemma 4.2, the *to-low* edge of v is a potential generator with the potential path at v being $low(v) \rightsquigarrow 2nd\text{-}low(v)$. An entry $[(v, to\text{-}low(v)), low(v) \rightsquigarrow 2nd\text{-}low(v)]$ is thus pushed onto $stack(v)$.

On the other hand, if v has a *to-low* child, then $stack(v)$ is initialized to $stack(to\text{-}low(v))$. This is because by Lemma 3.6, any edge lying in $T(v)$ that forms a cut-pair with an edge outside $T(v)$ must be an edge on the *to-low* path of v. It follows that the edge is either the *to-low* edge of v (i.e. $(v, to\text{-}low(v))$) or an edge on the *to-low* path of $to\text{-}low(v)$. The stacks of the other children can thus be discarded. Now, let $h = dfs(q)$ if there is a top entry $[(*, p \rightsquigarrow q)]$ on $stack(v)$, and let $h = lowpt(v)$

otherwise. If $h < 2nd\text{-}lowpt(v)$, then by Lemma 4.2 or Lemma 4.3(i), the edge $(v, to\text{-}low(v))$ is a potential generator with $w_h \rightsquigarrow 2nd\text{-}low(v)$ being its potential path at v, where $dfs(w_h) = h$. As a result, an entry $[(v, to\text{-}low(v)), w_h \rightsquigarrow 2nd\text{-}low(v)]$ is pushed onto $stack(v)$ (Fig. 3(iii)).

Otherwise, let $stack(v)$ be consisting of $\{[(x_j, y_j), p_j \rightsquigarrow q_j] \mid 1 \leqslant j \leqslant k\}$ with $[(x_k, y_k), p_k \rightsquigarrow q_k]$ being the top entry. From the definition of $stack(v)$, we have: $dfs(x_j) \geqslant dfs(y_{j+1})$, $q_j = p_{j+1}$, $1 \leqslant j < k$, and $dfs(q_k) \leqslant dfs(v) \leqslant dfs(x_k)$. Let m be the smallest index such that $2nd\text{-}lowpt(v) \leqslant dfs(p_m)$. Then the entries $[(x_j, y_j), p_j \rightsquigarrow q_j]$, $1 \leqslant j \leqslant m$, are popped out of the stack, because none of these (x_j, y_j)'s could generate new cut-pairs owing to Lemma 4.3(iii). If the resulting $stack(v)$ is non-empty, then $[(x_{m-1}, y_{m-1}), p_{m-1} \rightsquigarrow q_{m-1}]$ becomes the top entry and if $dfs(p_{m-1}) < 2nd\text{-}lowpt(v) < dfs(q_{m-1})$, the path $p_{m-1} \rightsquigarrow q_{m-1}$ in the top entry is replaced by $p_{m-1} \rightsquigarrow 2nd\text{-}low(v)$ because the latter is the potential path of (x_{m-1}, y_{m-1}) at v owing to Lemma 4.3(ii) (Fig. 3(iv)).

Each incoming back-edge, (u, v), of v is then examined and all the entries $[(x, y), p \rightsquigarrow q]$ on $stack(v)$ for which (x, y) is a tree-edge and u is a descendant of y are popped out of the stack (Fig. 3(v)). This is because none of them could generate new cut-pairs owing to Lemma 4.4. The depth-first search traversal then backtracks to the parent of v.

When the traversal returns to r, all edges that are marked as cut-edge form the set E_{cut}.

5. A simple linear-time sequential algorithm

The algorithm is presented as Algorithm 3-edge-connectivity below. It uses the procedure Find-cut-pairs(v) to carry out the depth-first search traversal.

Algorithm 3-edge-connectivity
Input: A connected bridgeless graph $G = (V, E)$ represented by the adjacency lists, $L[v]$, $\forall v \in V$;
Output: The set of cut-edges E_{cut} and the graph $G - E_{\text{cut}}$.
begin
$dfs \leftarrow 1$; mark all vertices as "unvisited";
call Find-cut-pairs(r); /* Determine E_{cut} */
end.

Procedure Find-cut-pairs(v);
begin /* Initialization */
Mark v as "*visited*";
$dfs(v) \leftarrow dfs$; $dfs \leftarrow dfs + 1$; /* assign a depth-first search number to v */
$nd(v) \leftarrow 1$; /* initialize, $nd(v)$, the number of descendants of v in T */
$lowpt(v) \leftarrow dfs(v)$; $low(v) \leftarrow v$; /* initialize $lowpt(v), low(v)$ */
$2nd\text{-}lowpt(v) \leftarrow dfs(v)$; $2nd\text{-}low(v) \leftarrow v$; /* initialize $2nd\text{-}lowpt(v), 2nd\text{-}low(v)$*/

1. **for** (*each vertex* $w \in L[v]$) **do**
   1.1 **if** (w *is marked unvisited*) **then**
      call Find-cut-pairs(w);
      /* Let $[(x, y), p \rightsquigarrow q]$ denote the top element of $stack(w)$, if non-empty.*/
      1.1.1 **if** (($stack(w)$ is non-empty) $\wedge (w = q)$) **then** pop $stack(w)$;
         /* a cut-pair $\{(x, y), (v, w)\}$ has been found */
         mark edges (v, w) and (x, y) as cut-edges;
         if $(v \neq p)$ **then** push $[(x, y), p \rightsquigarrow v]$ onto $stack(w)$;
      fi;
      $nd(v) \leftarrow nd(v) + nd(w)$; /* to be used in testing ancestor/descendant relationship */
      1.1.2 **if** ($lowpt(w) < lowpt(v)$) **then** /* w becomes the new $to\text{-}low(v)$ */
         $2nd\text{-}lowpt(v) \leftarrow lowpt(v)$; $lowpt(v) \leftarrow lowpt(w)$;
         $2nd\text{-}low(v) \leftarrow low(v)$; $low(v) \leftarrow low(w)$;
         $stack(v) \leftarrow stack(w)$; $to\text{-}low(v) \leftarrow w$
      1.1.3 **else if** ($lowpt(w) < 2nd\text{-}lowpt(v)$) **then** /* update $2nd\text{-}low(v), 2nd\text{-}lowpt(v)$ */
         $2nd\text{-}lowpt(v) \leftarrow lowpt(w)$; $2nd\text{-}low(v) \leftarrow low(w)$;
         empty $stack(w)$
   1.2 **else if** ((v, w) *is an outgoing back-edge of* v) **then**
      1.2.1 **if** ($dfs(w) \leqslant lowpt(v)$) **then**
         $2nd\text{-}lowpt(v) \leftarrow lowpt(v)$; $lowpt(v) \leftarrow dfs(w)$;
         $2nd\text{-}low(v) \leftarrow low(v)$; $low(v) \leftarrow w$;
         empty $stack(v)$; $to\text{-}low(v) \leftarrow w$;

else if ($dfs(w) < 2nd\text{-}lowpt(v)$) **then**
$2nd\text{-}lowpt(v) \leftarrow dfs(w)$; $2nd\text{-}low(v) \leftarrow w$;

2. /* In the following steps, the top element of $stack(v)$ is always denoted by $[(x, y), p \rightsquigarrow q]$ */
2.1 **if** ($stack(v)$ *is empty*) **then**
2.1.1 **if** ($2nd\text{-}lowpt(v) > lowpt(v)$) **then**
push $[(v, to\text{-}low(v)), low(v) \rightsquigarrow 2nd\text{-}low(v)]$ onto $stack(v)$
2.2 **else if** ($2nd\text{-}lowpt(v) > dfs(q)$) **then**
2.2.1 push $[(v, to\text{-}low(v)), q \rightsquigarrow 2nd\text{-}low(v)]$ onto $stack(v)$
2.2.2 **else while** (($stack(v)$ *is non-empty*) $\wedge$ ($2nd\text{-}lowpt(v) \leqslant dfs(p)$)) **do** pop $stack(v)$;
if (($stack(v)$ *is non-empty*) $\wedge$ ($2nd\text{-}lowpt(v) < dfs(q)$)) **then**
pop $stack(v)$;
push $[(x, y), p \rightsquigarrow 2nd\text{-}low(v)]$ onto $stack(v)$

3. **for** (*each* $u \in L[v]$ *such that* (u, v) *is an incoming back-edge of* v) **do**
while (($stack(v)$ *is non-empty*) $\wedge$ ((x, y) *is a tree-edge*) $\wedge$ (u *is a descendant of* y))
do pop $stack(v)$;
end;

Lemma 5.1. *For every $v \in V$, upon returning from the procedure call Find-cut-pairs(v), if $\{(x, y), (s, t)\}$ is a cut-pair such that (x, y) is a generator and (x, y) lies in $T(v)$ but not (s, t), then there exists an entry $[\{(x, y), p \rightsquigarrow q\}]$ on $stack(v)$ such that $pre(q) \leqslant pre(v) \leqslant pre(x)$ and (s, t) lies on the path $p \rightsquigarrow q$.*

Proof. (By induction on the level of the vertices in the *DFS-tree*, T.)

First, consider the case where v is a leaf in T. Then (v, w) is a back-edge for every $w \in L[v]$. Therefore, w is *visited* when the edge (v, w) is examined at v. It follows that in Step 1, only statement 1.2.1 is executed for every $w \in L[v]$. In Statement 1.2.1, whenever $dfs(w) \leqslant lowpt(v)$, $dfs(w)$ correctly becomes the new $lowpt(v)$ value while the current $lowpt(v)$ value correctly becomes the new $2nd\text{-}lowpt(v)$ value. Moreover, $low(v), 2nd\text{-}low(v)$ and $to\text{-}low(v)$ are also correctly updated. Note that as $stack(v)$ is always empty in this case, the "empty $stack(v)$" statement is a dummy statement. Whenever $lowpt(v) < dfs(w) < 2nd\text{-}lowpt(v)$, $dfs(w)$ correctly becomes the new $2nd\text{-}lowpt(v)$ value and $2nd\text{-}low(v)$ is correctly updated to w. Since both $lowpt(v)$ and $2nd\text{-}lowpt(v)$ are correctly initialized to $dfs(v)$ initially, when execution of the **for** loop terminates, $low(v)$ and $2nd\text{-}low(v)$ are the vertices with the smallest and second smallest depth-first-search numbers, respectively, among all the vertices that are connected to v via a back-edge, and $lowpt(v), 2nd\text{-}lowpt(v)$ are their respective depth-first-search numbers. Moreover, the edge $(v, to\text{-}low(v))$ is the *to-low* edge of v and $stack(v)$ is empty.

It follows that in Statement 2, only Statement 2.1.1 is executed and if $2nd\text{-}lowpt(v) > lowpt(v)$, the entry $[(v, to\text{-}low(v)), low(v) \rightsquigarrow 2nd\text{-}low(v)]$ is pushed onto $stack(v)$. By Lemma 4.2, the $low(v)$-$2nd\text{-}low(v)$ tree-path contains all the edges that can form cut-pairs with the edge $(v, to\text{-}low(v))$. The entry $[(v, to\text{-}low(v)), low(v) \rightsquigarrow 2nd\text{-}low(v)]$ is thus correctly pushed onto the stack. Since there is no incoming back-edge of v, this entry remains on $stack(v)$ after Step 3. The lemma thus holds for this case.

Now, consider the case where v is an internal vertex of T. By the definition of $to\text{-}low(v)$ and Lemma 3.6, if $\{(x, y), (s, t)\}$ is a cut-pair such that (x, y) is a generator and (x, y) lies in $T(v)$ but not (s, t) (recall that a back-edge is considered as lying in $T(v)$ if one of its end-vertices is in $T(v)$), then either (x, y) is the *to-low edge* of v or (x, y) lies on the *to-low path* of $to\text{-}low(v)$. It follows that $stack(v)$ must be constructed based on $stack(to\text{-}low(v))$ if $to\text{-}low(v)$ is a child of v.

Therefore, for every $w \in L[v]$, if w is a child of v such that $lowpt(w) \geqslant lowpt(v)$, then $w \neq to\text{-}low(v)$. As a result, $stack(v)$ is correctly retained in Statement 1.1.3. Furthermore, if $lowpt(v) \leqslant lowpt(w) < 2nd\text{-}lowpt(v)$, then $2nd\text{-}lowpt(v)$ and $2nd\text{-}low(v)$ are correctly updated to $lowpt(w)$ and $low(w)$, respectively; if w is a child of v such that $lowpt(w) < lowpt(v)$, then vertex w is the potential $to\text{-}low(v)$, therefore $stack(v)$ and $to\text{-}low(v)$ are correctly updated to $stack(w)$ and w, respectively in Statement 1.1.2. Moreover, $lowpt(v)$, $2nd\text{-}lowpt(w)$, $low(v)$ and $2nd\text{-}low(w)$, are also correctly updated. On the other hand, if w is not a child of v (i.e. (v, w) is a back-edge) and $dfs(w) \leqslant lowpt(v)$, then w becomes $to\text{-}low(v)$ and $stack(v)$ is correctly emptied in Statement 1.2.1 because $to\text{-}low(v)(=w)$ is not a child of v and hence does not have a stack. Moreover, $lowpt(v)$, $2nd\text{-}lowpt(w)$, $low(v)$ and $2nd\text{-}low(w)$, are also correctly updated. If w is not a child of v but $lowpt(v) < dfs(w) < 2nd\text{-}lowpt(v)$, then $stack(v)$ is correctly retained because $w \neq to\text{-}low(v)$ and only $2nd\text{-}low(v)$ and $2nd\text{-}lowpt(v)$ are updated accordingly. Consequently, when execution of the **for** loop terminates, $stack(v)$ correctly inherited the stack of $to\text{-}low(v)$ if the latter is a child of v and is empty, otherwise. Moreover, $low(v)$, $lowpt(v)$, $2nd\text{-}low(v)$, etc., are correctly computed.

Now, suppose $to\text{-}low(v)$ is not a child of v. Then $stack(v)$ is empty. It follows that in Statement 2, Statement 2.1.1 is executed. The remaining part of the argument is same as that given earlier when v is a leaf in T.

Suppose $to\text{-}low(v)$ is a child of v. Then the procedure call Find-cut-pairs($to\text{-}low(v)$) was invoked during the execution of the **for** loop. By the induction hypothesis, upon returning from the procedure call, the following proposition holds for

to-low(v): "for every cut-pair $\{(x, y), (s, t)\}$ such that (x, y) is a generator and (x, y) lies in $T($*to-low*$(v))$ but not (s, t), there exists an entry $[(x, y), p \rightsquigarrow q]$ on *stack*$($*to-low*$(v))$ such that *dfs*$(q) \leqslant$ *dfs*$($*to-low*$(v)) \leqslant$ *dfs*(x) and (s, t) lies on the tree-path $p \rightsquigarrow q$." If *stack*$($*to-low*$(v))$ is non-empty, let $[(x, y), p \rightsquigarrow q]$ be the top entry on it. If *dfs*$(q) >$ *dfs*(v) (i.e. $w = q$), then after executing Statements 1.1.1, *dfs*$(q) \leqslant$ *dfs*(v). Furthermore, by the definition of *to-low*(v), Statement 1.1.2 is executed which results in assigning *stack*$($*to-low*$(v))$ to *stack*(v). Then *stack*(v) will remain unchanged until execution of the **for** loop terminates. Therefore, upon exiting the **for** loop, the proposition becomes: "for every cut-pair $\{(x, y), (s, t)\}$ such that (x, y) is a generator and (x, y) lies in $T($*to-low*$(v))$ but not (s, t), there exists an entry $[(x, y), p \rightsquigarrow q]$ on *stack*(v) such that *dfs*$(q) \leqslant$ *dfs*$(v) <$ *dfs*(x) and (s, t) lies on the tree-path $p \rightsquigarrow q$." If *stack*$($*to-low*$(v))$ is empty, then the proposition automatically holds.

Now, if $(v,$ *to-low*$(v))$ is not a generator, then the proposition remains valid after $T(v)$ replaces $T($*to-low*$(v))$ and the inequality is changed to *dfs*$(q) \leqslant$ *dfs*$(v) \leqslant$ *dfs*(x). The resulting proposition is that of the lemma.

On the other hand, if $(v,$ *to-low*$(v))$ is indeed a generator, then owing to Lemmas 3.2(ii), 4.2 and 4.3(i), we must have *lowpt*$(v) <$ 2*nd-lowpt*(v) if *stack*(v) is empty or *dfs*$(q) <$ 2*nd-lowpt*(v) if *stack*(v) is non-empty and $p \rightsquigarrow q$ is the path-component of its top entry. It follows that an entry of the form $[(v,$ *to-low*$(v))$, *low*$(v) \rightsquigarrow$ 2*nd-low*$(v)]$ or $[(v,$ *to-low*$(v)), q \rightsquigarrow$ 2*nd-low*$(v)]$ is pushed onto *stack*(v) in Statement 2.1.1 or 2.2.1. In either case, we have an entry $[(v,$ *to-low*$(v)), \alpha \rightsquigarrow \beta]$ on *stack*(v) such that the path $\alpha \rightsquigarrow \beta$ contains all the edges outside $T(v)$ that could form cut-pairs with $(v,$ *to-low*$(v))$. This statement combined with the above proposition give rise to the proposition in the lemma. Finally, as Statement 2.2.2 and Step 3 remove entries from *stack*(v) that are known not to be able to generate cut-pair or shorten potential paths by eliminating edges on them that will not form cut-pairs with the corresponding potential generator owing to Lemmas 4.3(ii),(iii) and 4.4, they have no effect on the proposition. The lemma thus follows. □

Lemma 5.2. *For every $v \in V$, upon returning from the procedure call Find-cut-pairs(v), all cut-pairs and only those cut-pairs in $T(v)$ in which one cut-edge is a generator have been reported during the procedure call.*

Proof. (By induction on the level of the vertices in the *DFS-tree*, T.)

First, consider the case where v is a leaf in T. Since w is *visited*, $\forall w \in L[v]$, Statement 1.1 is skipped for every $w \in L[v]$. As a result, no cut-pair lying in $T(v)$ is reported. This is correct because the depth of $T(v)$ is 'zero' implying there is no cut-pair in $T(v)$.

Next, consider the case where v is an internal vertex of T. Let $w \in L[v]$. If w is not a child of v in T, then as with the above case, Statement 1.1 is skipped and no cut-pairs in $T(v)$ containing (v, w) is reported (note that the back-edge (v, w) is considered as an edge in $T(v)$ because v is a vertex in $T(v)$). This is correct because by Lemma 3.1(i), any edge that forms a cut-pair with the back-edge (v, w) must lie on the w–v tree-path which is outside $T(v)$.

Suppose w is a child of v in T. Then the procedure call Find-cut-pairs(w) is invoked. Upon returning from the procedure call, by the induction hypothesis, all cut-pairs and only those cut-pairs in $T(w)$ in which one cut-edge is a generator have been reported during the procedure call. Moreover, if (v, w) forms a cut-pair with some generator in $T(w)$, then, by Lemma 5.1, there exists an entry $[(x, y), p \rightsquigarrow q]$ on *stack*(w) such that *dfs*$(q) \leqslant$ *dfs*$(w) \leqslant$ *dfs*(x) and (v, w) lies on the path $p \rightsquigarrow q$. But (v, w) lying on $p \rightsquigarrow q$ implies that *dfs*$(w) \leqslant$ *dfs*(q). It follows that *dfs*$(w) =$ *dfs*(q). As a result, $w = q$. Owing to the nesting structure of the entries in *stack*(w), the entry $[(x, y), p \rightsquigarrow q]$ must be the top entry of *stack*(w). As a result, the condition in the **if** statement (Statement 1.1.1) is satisfied and the cut-pair $\{(x, y), (v, w)\}$ is correctly reported.

On the other hand, suppose (v, w) does not form a cut-pair with any edge in $T(w)$. Let (x, y) be an edge lying in $T(w)$ such that in the course of executing Find-cut-pairs(w), an entry $[(x, y), p \rightsquigarrow q]$ is pushed onto a stack with the path $p \rightsquigarrow q$ containing (v, w).

(i) Suppose (x, y) is a tree-edge. If (x, y) does not lie on the *to-low path* of w, then let z be the closest ancestor of x lying on the *to-low path* of w. Then during the execution of Find-cut-pairs(z), the entry $[(x, y), p \rightsquigarrow q]$ (note that q could have been changed as the p–q tree-path could have been shorten by Statement 2.2.2 earlier) is eliminated when a stack containing it is destroyed by Statement 1.1.2, or 1.1.3, or 1.2.1 owing to the stack does not belong to *to-low*(z). If (x, y) does lie on the *to-low path* of w, then as $\{(x, y)(v, w)\}$ is not a cut-pair, by Lemma 3.2(ii), there exists a back-edge (s, t) such that either s is a descendant of w but not of y and t is an ancestor of v, or s is a descendant of y and t is an ancestor of x but not of v.

In the former case, let z be the lowest-common-ancestor of s and x. When the depth-first search backtracks to z, the entry $[(x, y), p \rightsquigarrow q]$ or a section of $p \rightsquigarrow q$ containing (v, w) will be eliminated by Statement 2.2.2. In the latter case, when the depth-first search backtracks to vertex t, the entry $[(x, y), p \rightsquigarrow q]$ will be eliminated in Step 3. Therefore, upon returning from the procedure call Find-cut-pairs(w), the condition in the **if** statement (Statement 1.1.1) must be false. As a result, no cut-pair containing (v, w) and a generator in $T(w)$ is reported.

(ii) Suppose (x, y) is a back-edge. The argument is similar to case (i) except the condition in Lemma 3.2(i) is used. The lemma thus follows. □

Theorem 5.3. *Algorithm* 3*-edge-connectivity correctly determines E_{cut} and hence the graph G–E_{cut}.*

Proof. Immediate from Lemma 5.2. □

Theorem 5.4. *Algorithm* 3*-edge-connectivity runs in $O(|E|)$ time and space.*

Proof. First, we shall explain how to mark a cut-edge in $O(1)$ time. For every edge (v, w), the node of vertex w in $L[v]$ contains a pointer pointing at the node of vertex v in $L[w]$ and vice versa. These pointers can be easily created when the adjacency lists $L[v], v \in V$, are created. Whenever the instruction "*to-low*$(v) \leftarrow w$" is executed in Statement 1.1.2 or 1.2.1, a pointer pointing at the node of w, and hence of *to-low*(v), in $L[v]$ is maintained. When a vertex v creates a new entry $[(v, \textit{low-to}(v)), p \rightsquigarrow q]$ for its stack in statement 2.1.1 or 2.2.1, the pointer pointing at the node of *to-low*(v) in $L[v]$ is stored along with the edge $(v, \textit{low-to}(v))$ on the stack. Therefore, when a cut-pair is detected in Statement 1.1.1 and an entry $[(x, y), p \rightsquigarrow q]$ is popped out of *stack*(w), the pointer pointing at the node of y in $L[x]$ can be retrieved (note that *to-low*$(x) = y$). By using this pointer and the pointer kept in that node of y which points at the node of x in $L[y]$, the two nodes can be retrieved and marked to indicate that the edge (x, y) is a cut-edge. Since the node of w in $L[v]$ is being examined, by using its pointer pointing at the node of v in $L[w]$, the latter can be retrieved. Both nodes can then be marked to indicate that the edge (v, w) is a cut-edge.

The first statement of Algorithm 3-edge-connectivity clearly takes $O(|V|)$ time.

For each vertex $v \in V$, the initialization steps in procedure Find-cut-pairs takes $O(1)$ time. In Step 1, excluding the recursive calls of Find-cut-pairs (which will be charged to the child vertices), every statement in the **for** loop takes $O(1)$ time. The total time spent on these statements is thus $\sum_{w \in L[v]} O(1) = O(\textit{deg}(v))$, where $\textit{deg}(v)$ is the degree of v in G.

In Step 2, excluding the **while** loop (Statement 2.2.2), every statement takes $O(1)$ time. In Step 3, testing the condition in the **for** loop for each $u \in L[v]$ takes $O(1)$ time. The **for** loop (excluding the nested **while** loop) thus takes a total of $\sum_{u \in L[v]} O(1) = O(\textit{deg}(v))$ time. The **while** loop in Statement 2.2.2 takes O(1) time for each iteration. The **while** loop in Step 3 also takes O(1) time for each iteration owing to Lemma 3.4. Let $t(v)$ be the total number of entries popped out of *stack*(v) by these two **while** loops. Then the two loops take a total of $O(t(v))$ time. The execution of Find-cut-pairs(r) thus takes a total of $\sum_{v \in V} [O(1) + O(\textit{deg}(v)) + O(1) + O(\textit{deg}(v)) + O(t(v))] = O(|E|) + \sum_{v \in V} O(t(v))$ time.

Since at most one stack entry is created for every edge and once the entry is popped out of a stack by Statement 2.2.2 or Step 3, it will not be pushed back to any stack later on, therefore every entry popped out by the two **while** loops in Statement 2.2.2 and Step 3, respectively, corresponds to a distinct edge. This implies that $\sum_{v \in V} t(v) \leqslant |E|$. It follows that $\sum_{v \in V} O(t(v)) = O(|E|)$. The lemma thus follows. The space complexity is easily verified. □

6. Computing the 3-edge-connected components

It turns out that the connected components of the graph $G - E_{\text{cut}}$ do not always correspond to the 3-edge-connected components of G. For instance, consider the complete bipartite graph $K_{2,3}$. The two vertices of degree 3 in the graph are 3-edge-connected to each other as there are three edge-disjoint paths connecting them. Each of the three paths consists of the two edges incident on one of the three remaining vertices that are of degree 2. However, as the two edges on each of these paths form a cut-pair (their removal separates their common end-vertex from the remaining vertices), all the edges in the graph are thus cut-edges. As a result the graph $K_{2,3} - E_{\text{cut}}$ is an edgeless graph and the two vertices of degree 3 in $K_{2,3}$ thus belong to different connected components in $K_{2,3} - E_{\text{cut}}$. To alleviate this problem, an edge connecting the two vertices can be added to $K_{2,3} - E_{\text{cut}}$. The connected components of the resulting graph will then correspond to the 3-edge-connected components of $K_{2,3}$.

In general, let $e' = (x, y)$ be any generator that is a tree-edge. Owing to Lemma 3.1, every cut-edge that forms a cut-pair with e' lies on the path connecting x with the root of T. Let $e = (v, w)$ be the edge closest to the root that forms a cut-pair with e'. A new edge (v, y) is added to $G - E_{\text{cut}}$. Let the resulting graph be $\overline{G}$.

Lemma 6.1. *(See [10].) Let $U \subseteq V$. U is a 3-edge-connected component of G if and only if U is the vertex set of a connected component of $\overline{G}$.*

Algorithm 3-edge-components can be easily modified to construct the graph $\overline{G}$ as follows.

First, recall that the input graph G is represented by the adjacency lists data structure. What is required is a new array *newlink*$(y), y \in V$, such that *newlink*(y) is the farthest ancestor of y of whom one of the child edge form a cut-pair with the parent edge of y. Initially, *newlink*(y) is set to y. In executing Statement 1.1.1, whenever the generator (x, y) is a tree-edge, *newlink*(y) is updated to v to record vertex v as the farthest ancestor (up to that point of time) that has a child edge forming a cut-pair with (x, y). Therefore, when the depth-first search terminates at the root of T, *newlink*(y) is the desired vertex if *newlink*$(y) \neq y$.

To create the adjacency lists for $\overline{G}$, the adjacency lists of G is scanned to remove all the *marked* nodes that correspond to the cut-edges. Then for each $y \in V$ such that *newlink*$(y) \neq y$, a node *newlink*(y) is added to the adjacency list of y while a node y is added to the adjacency list of *newlink*(y).

Since every tree-edge can form a cut-pair with at most one generator, the total number of updates performed on the array *newlink* is thus $O(|V|)$ time. Furthermore, initializing the array take $O(|V|)$ time; converting the adjacency list of G to that of $G - E_{\text{cut}}$ takes $O(|E|)$ time and adding the new edges to the adjacency list of $G - E_{\text{cut}}$ takes $O(|V|)$ time. It thus takes a total of $O(|E|)$ time to create the adjacency lists of $\overline{G}$. Since the array *newlink* requires only $O(|V|)$ space, the space complexity remains as $O(|E|)$. Finally, determining the vertex set of the connected components of $\overline{G}$ can easily be done in $O(|E|)$ time and space. We thus have:

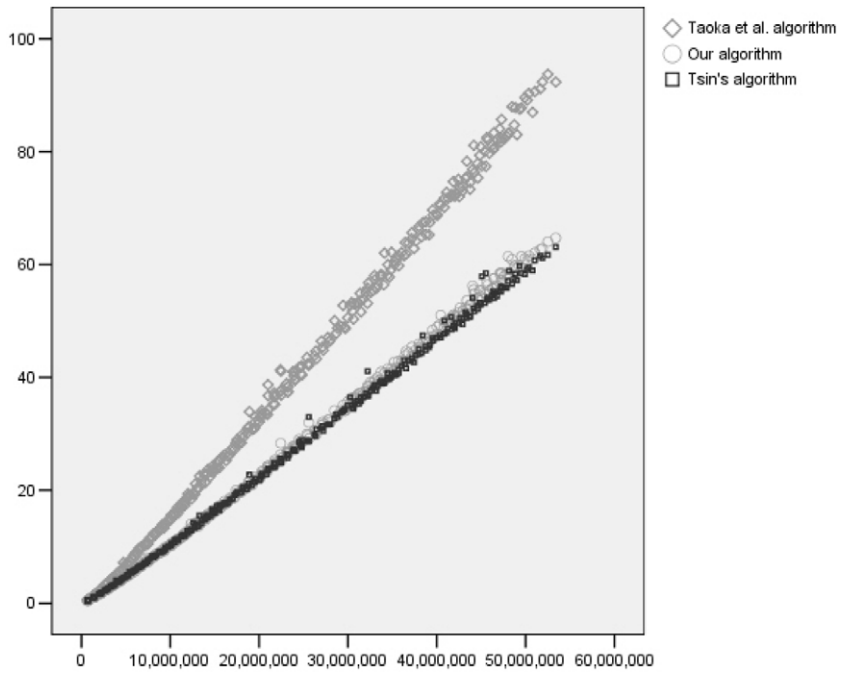


Fig. 4. Determining the 3-edge-connected components (sparse graphs).

Theorem 6.2. *The* 3*-edge-connected component of G can be determined in* $O(|E|)$ *time and space.*

7. Experimental results

Although the algorithm of Galil et al. [2] is theoretically as elegant as the others, it uses reduction and is based on the rather complicated 3-vertex-connectivity algorithm of Hopcroft and Tarjan [3]. Similarly, the algorithm of Nagamochi et al. [6] is complicated as it performs multiple passes over the graph and uses three different types of transformation to transform the graph. We therefore performed the experimental study on the algorithm of Taoka et al. [10], Tsin's algorithm [13] and our algorithm.

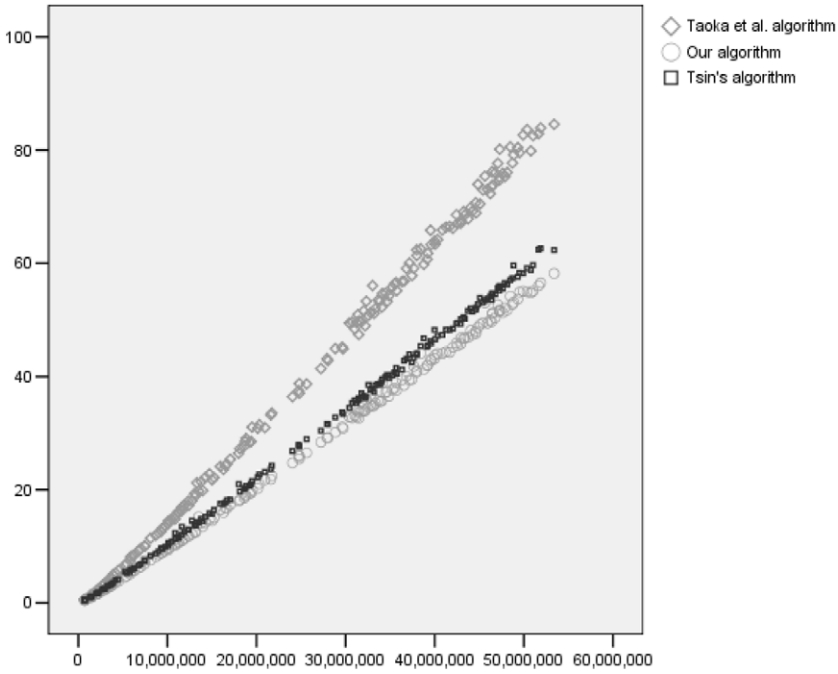
All the tests were conducted on a Dell Precision Workstation 650 with an 3.2 GHz Intel/Xeon dual processor, a 512 KB L2 cache and a 4 GB memory. The operating system is Fedora core Linux 2.6.12. The programs are written in C.

The input graphs were randomly generated. The number of edges in the graph ranges from 650,000 to 66 millions. Two sets of graphs were generated each with roughly 300 graphs. The first set consists of sparse graphs while the second set consists of dense graphs. All of the graphs are 2-edge-connected as is assumed by the algorithms.

Four sets of tests were performed. In the first set, the algorithms were tested on generating the 3-edge-connected components. In the second set, the algorithms were tested on identifying graphs that are 3-edge-connected (the *yes* input instances). In the third set, the algorithms were tested on identifying graphs that are *not* 3-edge-connected (the *no* input instances). In the last set, the algorithms were tested on reporting cut-pairs for those graphs that are *not* 3-edge-connected. The result of the tests are depicted in Figs. 4 to 11.

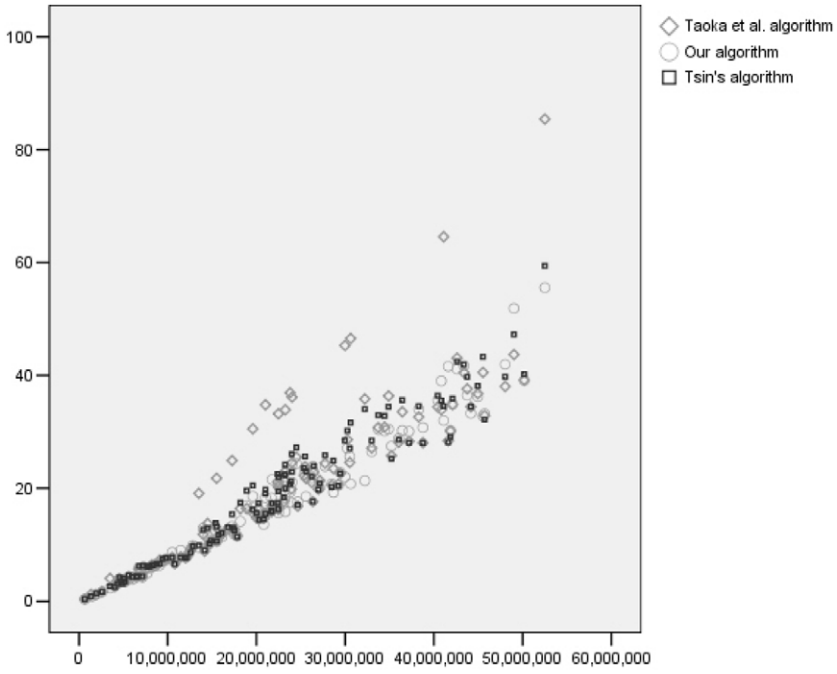
For sparse graphs (Figs. 4 to 7), it is obvious that the algorithm of Taoka et al. runs the slowest among the three algorithms. The only exception is the test for identifying graphs that are not 3-edge-connected (Fig. 6). In this test, since the algorithms can terminate their execution immediately after finding the first cut-pair, if the algorithm of Taoka et al. is able to find a type-1 cut-pair, then it runs just as fast as the other two algorithms as it does not need to make a second pass over the input graph. Otherwise, it would have to perform the time-consuming second and third steps to find a type-2 cut-pair. Our algorithm runs faster that Tsin's algorithm except for the test for determining all the 3-edge-connected components (Fig. 4). This is because our algorithm has to make one pass over the input graph to determine all the cut-pairs and then another pass over a modified input graph to determine all the 3-edge-connected components. However, the difference between the performances of the two algorithms is not very significant. This observation suggests that even though Tsin's algorithm is a simple one-pass algorithm, the operations it must perform on the adjacency lists in order to gradually transform the given graph into the desired edgeless graph could induce substantial overhead during run-time.

For dense graphs (Figs. 8 to 11), even though our algorithm appears to run the fastest while the algorithm of Taoka et al. appears to be the slowest in most cases, the difference between the performances of the three algorithms is not as significant as the case for sparse graphs. Furthermore, with the exception of the test for determining the 3-edge-connected



X-axis: Input size ($|E| + |V|$), Y-axis: Execution time (Second)

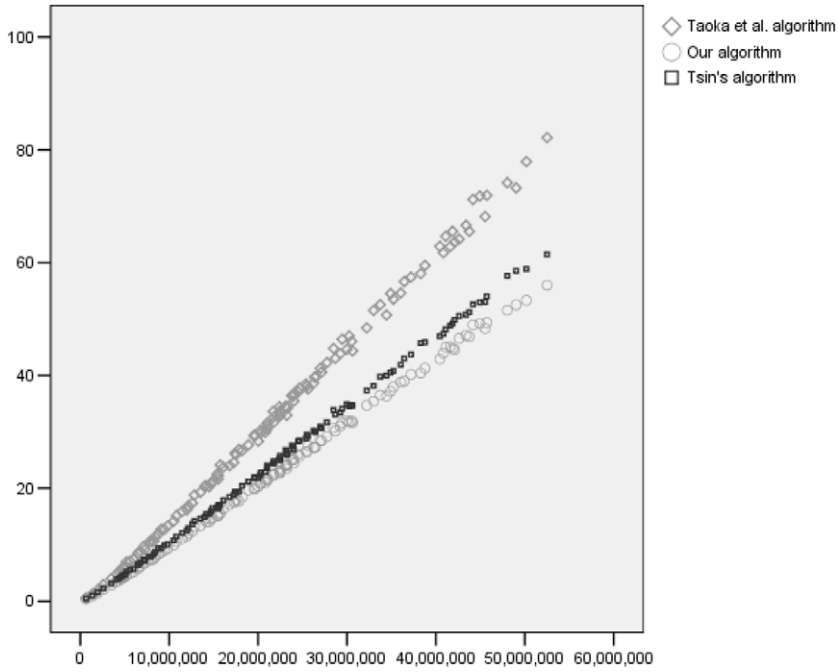
Fig. 5. Determining 3-edge-connectivity; only Yes input instances (sparse graphs).



X-axis: Input size ($|E| + |V|$), Y-axis: Execution time (Second)

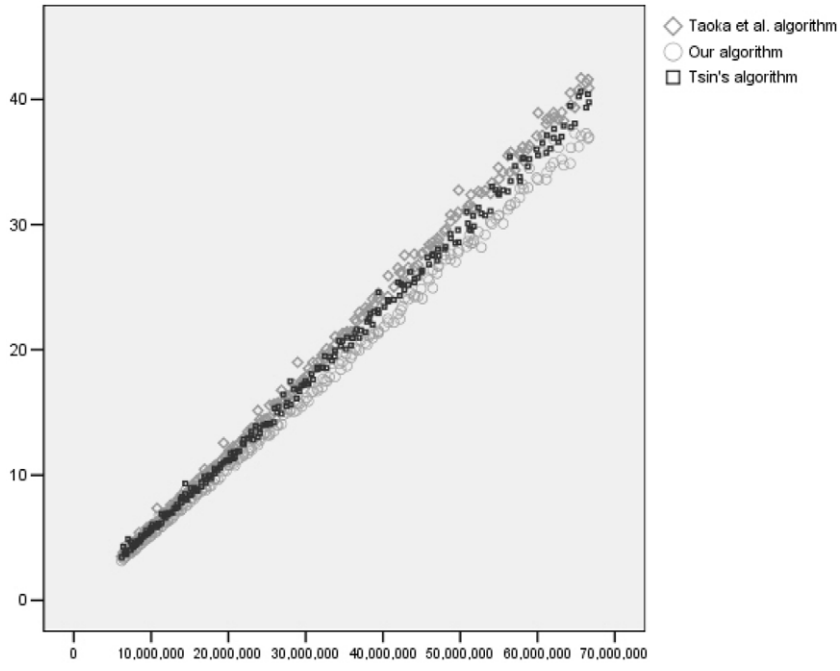
Fig. 6. Determining 3-edge-connectivity; only No input instances (sparse graphs).

components (Fig. 8), Tsin's algorithm has no obvious advantage over the algorithm of Taoka et al. This observation again suggests that transforming the input graph into the desired edgeless graph in Tsin's algorithm is rather time-consuming during run-time.



X-axis: Input size ($|E| + |V|$), Y-axis: Execution time (Second)

Fig. 7. Determining cut-pairs on No input instances (sparse graphs).

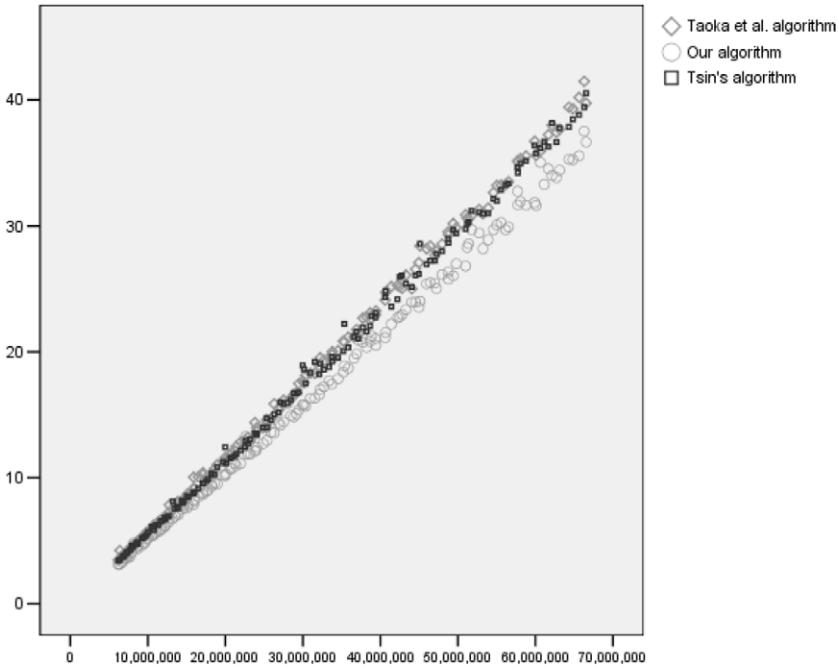


X-axis: Input size ($|E| + |V|$), Y-axis: Execution time (Second)

Fig. 8. Determining 3-edge-connected components (dense graphs).

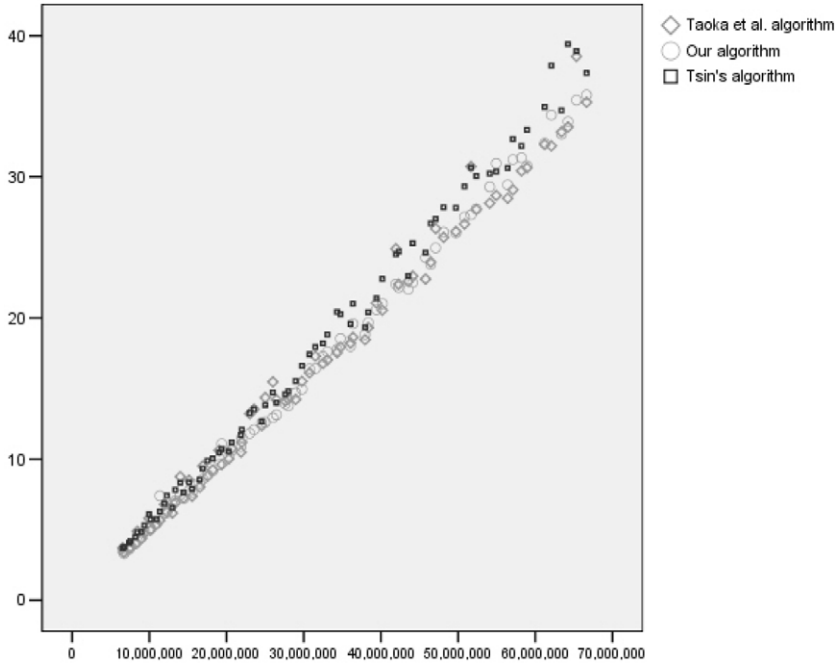
8. Concluding remarks

We have presented a new and simple algorithm for 3-edge-connectivity. The algorithm runs in linear time and space. Experimental results show that it outperforms all the previously known linear-time algorithms for 3-edge-connectivity if



X-axis: Input size ($|E| + |V|$), Y-axis: Execution time (Second)

Fig. 9. Determining 3-edge-connectivity; only Yes input instances (dense graphs).

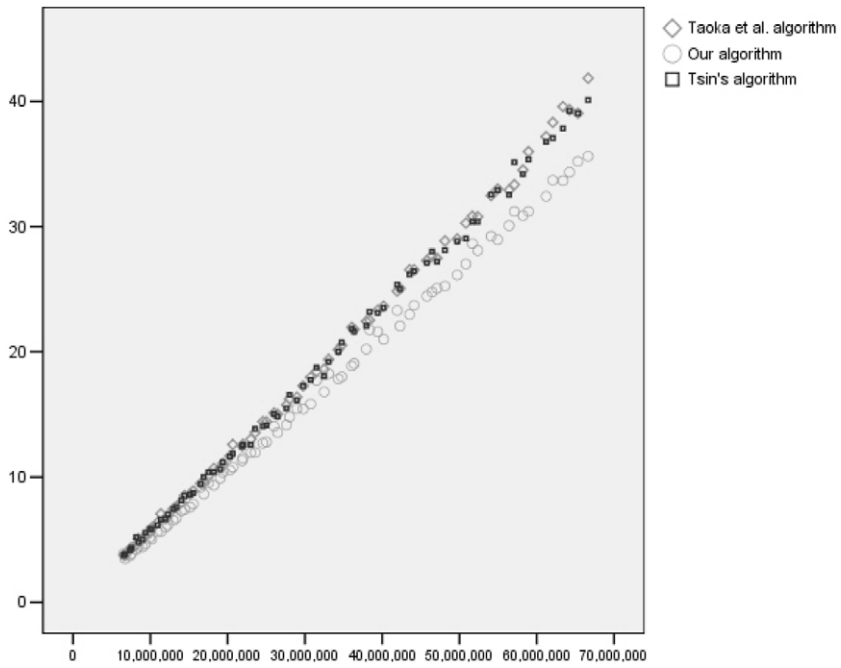


X-axis: Input size ($|E| + |V|$), Y-axis: Execution time (Second)

Fig. 10. Determining 3-edge-connectivity; only No input instances (dense graphs).

the objective is to determine if a given graph is 3-edge-connected or to generate all the cut-pairs. If the 3-edge-connected components are to be generated as well, it only runs slightly slower than the algorithm of Tsin [13] for sparse graphs.

Finally, it is worth noting that since depth-first search explores a graph one bridge-connected component at a time in a bottom-up manner, our algorithm can be easily modified so that it would work on any graph directly without having to decompose the graph into bridge-connected components first.



X-axis: Input size ($|E| + |V|$), Y-axis: Execution time (Second)

Fig. 11. Determining cut-pairs on No input instances (dense graphs).

References

[1] H. Gabow, Path-based depth-first search for strong and biconnected components, Inform. Process. Lett. 74 (2000) 107–114.
[2] Z. Galil, G.F. Italiano, Reducing edge-connectivity to vertex-connectivity, SIGACT News 22 (1991) 57–61.
[3] J.E. Hopcroft, R.E. Tarjan, Dividing a graph into triconnected components, SIAM J. Comput. 2 (3) (1973) 135–158.
[4] A. Kanevsky, V. Ramachandran, Improved algorithm for graph four-connectivity, J. Comput. System Sci. 42 (1991) 288–306.
[5] D. Matula, Determining edge-connectivity in o(mn) time, in: Proceedings of 28th Annual Symposium on Foundations of Computer Science, 1987, pp. 249–251.
[6] H. Nagamochi, T. Ibaraki, A linear time algorithm for computing 3-edge-connected components in a multigraph, Japan J. Indust. Appl. Math. 8 (1992) 163–180.
[7] N. Nakanishi, Graph Theory and Feynman Integrals, Gordon and Bridge Science Publishers, New York, 1971.
[8] J. Negele, H. Orland, Quantum Many-Particle Systems, Addison Wesley, Redwood City, CA, 1988.
[9] P. Shaw, Private communication, University of Newcastle, Newcastle, Australia, 2006.
[10] S. Taoka, T. Watanabe, K. Onaga, A linear time algorithm for computing all 3-edge-connected components of a multigraph, IEICE Trans. Fundamentals E75 3 (1992) 410–424.
[11] R.E. Tarjan, Depth-first search and linear graph algorithms, SIAM J. Comput. 1 (1972) 146–160.
[12] R.E. Tarjan, A note on finding the bridges of a graph, Inform. Process. Lett. 2 (2) (1974) 160–161.
[13] Y.H. Tsin, A simple 3-edge-connected component algorithm, Theory Comput. Syst. 40 (2) (2007) 125–142.
[14] Y.H. Tsin, F. Chin, A general program scheme for finding bridges, Inform. Process. Lett. 17 (1983) 269–272.