Washington University in St. Louis

Washington University Open Scholarship

All Computer Science and Engineering Research

Computer Science and Engineering

Report Number: wucse-2009-6

2009

A Simple Algorithm For Triconnectivity of a Multigraph

Abusayeed Saifullah and Alper Ungor

Vertex-connectivity and edge-connectivity represent the extent to which a graph is connected. Study of these key properties of graphs plays an important role in varieties of computer science applications. Recent years have witnessed a number of linear time 3-edge-connectivity algorithms - with increasing simplicity. In contrast, the state-of-the-art algorithm for 3-vertex-connectivity due to Hopcroft and Tarjan lacks the simplicity in the sense of ease of implementation as well as the number of passes over the graph although its time and space complexity is theoretically linear. In this paper, we propose a linear time reduction from 3-vertex-connectivity to 3-edge- connectivity... **Read complete abstract on page 2.**

Follow this and additional works at: https://openscholarship.wustl.edu/cse_research

Part of the Computer Engineering Commons, and the Computer Sciences Commons

Recommended Citation

Saifullah, Abusayeed and Ungor, Alper, "A Simple Algorithm For Triconnectivity of a Multigraph" Report Number: wucse-2009-6 (2009). *All Computer Science and Engineering Research.*
https://openscholarship.wustl.edu/cse_research/20

Department of Computer Science & Engineering - Washington University in St. Louis
Campus Box 1045 - St. Louis, MO - 63130 - ph: (314) 935-6160.

This technical report is available at Washington University Open Scholarship: https://openscholarship.wustl.edu/cse_research/20

A Simple Algorithm For Triconnectivity of a Multigraph

Abusayeed Saifullah and Alper Ungor

Complete Abstract:

Vertex-connectivity and edge-connectivity represent the extent to which a graph is connected. Study of these key properties of graphs plays an important role in varieties of computer science applications. Recent years have witnessed a number of linear time 3-edge-connectivity algorithms - with increasing simplicity. In contrast, the state-of-the-art algorithm for 3-vertex-connectivity due to Hopcroft and Tarjan lacks the simplicity in the sense of ease of implementation as well as the number of passes over the graph although its time and space complexity is theoretically linear. In this paper, we propose a linear time reduction from 3-vertex-connectivity to 3-edge- connectivity of a multigraph. This reduction was previously unknown, while the reduction in the opposite direction already exists. We apply an existing linear time 3-edge-connectivity algorithm on the reduced graph for solving the 3-vertex-connectivity of the original graph. Hence, for a graph with V vertices and E edges, the proposed reduction turns into an O(|V | +|E|) time and space algorithm for 3-vertex-connectivity while enjoying the simplicity of the 3-edge-connectivity algorithms.

Department of Computer Science & Engineering

Washington University in St. Louis
School of Engineering & Applied Science

2009-6

A Simple Algorithm For Triconnectivity of a Multigraph

Authors: Abusayeed Saifullah and Alper Ungor

Corresponding Author: saifullaha@cse.wustl.edu

Web Page: http://www.cse.wustl.edu/~saifullaha

Abstract: Vertex-connectivity and edge-connectivity represent the extent to which a graph is connected. Study of these key properties of graphs plays an important role in varieties of computer science applications. Recent years have witnessed a number of linear time 3-edge-connectivity algorithms - with increasing simplicity. In contrast, the state-of-the-art algorithm for 3-vertex-connectivity due to Hopcroft and Tarjan lacks the simplicity in the sense of ease of implementation as well as the number of passes over the graph although its time and space complexity is theoretically linear. In this paper, we propose a linear time reduction from 3-vertex-connectivity to 3-edge- connectivity of a multigraph. This reduction was previously unknown, while the reduction in the opposite direction already exists. We apply an existing linear time 3-edge-connectivity algorithm on the reduced graph for solving the 3-vertex-connectivity of the original graph. Hence, for a graph with V vertices and E edges, the proposed reduction turns into an O(|V | +|E|) time and space algorithm for 3-vertex-connectivity while enjoying the simplicity of the 3-edge-connectivity algorithms.

Type of Report: Other

Department of Computer Science & Engineering - Washington University in St. Louis
Campus Box 1045 - St. Louis, MO - 63130 - ph: (314) 935-6160

Reduction From 3-Vertex-Connectivity to 3-Edge-Connectivity

Abusayeed M. Saifullah[1] and Alper Üngör[2]

[1] Computer Science & Engineering
Washington University, St. Louis, MO 63130, USA
saifullaha@cse.wustl.edu
[2] Computer & Info. Science & Engineering
University of Florida, Gainesville, FL 32611, USA
ungor@cise.ufl.edu

Abstract. Vertex-connectivity and edge-connectivity represent the extent to which a graph is connected. Study of these key properties of graphs plays an important role in varieties of computer science applications. Recent years have witnessed a number of linear time 3-edge-connectivity algorithms - with increasing simplicity. In contrast, the state-of-the-art algorithm for 3-vertex-connectivity due to Hopcroft and Tarjan lacks the simplicity in the sense of ease of implementation as well as the number of passes over the graph although its time and space complexity is theoretically linear. In this paper, we propose a linear time reduction from 3-vertex-connectivity to 3-edge-connectivity of a multigraph. This reduction was previously unknown, while the reduction in the opposite direction already exists. We apply an existing linear time 3-edge-connectivity algorithm on the reduced graph for solving the 3-vertex-connectivity problem of the original graph. Hence, for a graph with $|V|$ vertices and $|E|$ edges, the proposed reduction turns into an $O(|V| + |E|)$ time and space algorithm for 3-vertex-connectivity while enjoying the simplicity of the 3-edge-connectivity algorithms.

1 Introduction

The vertex-connectivity and edge-connectivity problems are fundamental in graph theory. These properties measure the extent to which a graph is connected and have great importance in varieties of computer science applications [1–4, 12–14, 17–19, 21, 29]. The most direct applications arise in operation research for scheduling problems [2] and performance analysis of telecommunication systems and transportation networks [12, 14, 19]. An important criterion for performance analysis of a communication network is its reliability in the presence of link or node failures. Determining the highly connected subgraphs and partitioning the network into them is another important criterion for network analysis. Furthermore, when the communication links are expensive, the properties of graph connectivity play a vital role for minimizing the communication cost. The applications of graph connectivity also arise in irreducibility analysis of Feynman diagrams in quantum physics and chemistry [17, 18]; analysis of protein-protein networks obtained from microarray data in computational biology [21]; circuit lay-out problems [4]; and planarity testing [13].

Vertex-connectivity of a graph G is the smallest number of vertex deletions sufficient to disconnect G. Similarly, *edge-connectivity* of a graph G is the smallest number of edge deletions sufficient to disconnect G. A minimal set of nodes (edges, respectively) whose absence disconnects the graph is called a *separation* (*cut*, respectively). A separation containing only a pair of nodes (edges, respectively) is a *separation pair* (*cut pair*, respectively). A *3-vertex-connected* (*3-edge-connected*, respectively) graph contains no separation pair (cut pair, respectively). That is, in order to disconnect a 3-vertex-connected (3-edge-connected, respectively) graph, we have to remove at least three nodes (edges, respectively). The minimum vertex degree of a graph is an upper bound on both the edge-connectivity and the vertex-connectivity of the graph, since deleting all neighbors of a vertex of minimum degree disconnects that vertex from the rest of the graph. However, the vertex-connectivity is always no greater than the edge-connectivity, since deleting one vertex incident on each edge in a cut succeeds in disconnecting the graph. If a graph is not 3-edge-connected, then it is not 3-vertex-connected as well. In other words, a 3-vertex-connected graph is always 3-edge-connected. There is

no straightforward way to derive the vertex-connectivity of a graph from its edge-connectivity and vice versa.

Considering the degree of importance, much effort has been devoted to the study of graph connectivity problems in graph theory. For a graph with $|V|$ vertices and $|E|$ edges, Tarjan [24] proposed an $O(|V| + |E|)$ time algorithm for 2-vertex-connectivity and 2-edge-connectivity based on depth-first search. Gabow [6] also solved these problems in linear time by revisiting depth-first search from a different perspective - *the path-based view*. For 3-vertex-connectivity, the algorithm proposed by Hopcroft and Tarjan [11] runs in $O(|V| + |E|)$ time and space. The literature of graph theory is enriched with a number of $O(|V|+|E|)$ time and space algorithms for 3-edge-connectivity as well [7, 8, 16, 23, 26, 27]. Besides these, the linear time algorithm of Hsu et al. [22] can find a smallest set of edges whose addition triconnects an undirected graph. Based on edge or node insertions, the algorithm in [20] can answer, starting from an empty graph of n nodes, whether two nodes are triconnected in $O(n + m.\alpha(m, n))$ time, where m is the total number of queries and edge insertions, and α is the inverse Ackermann function. The triconnectivity problem was solved by Miller et al. [15] and Fussell et al. [5] for a parallel computer model.

The algorithm of Hopcroft and Tarjan [11] is currently, to the best of our knowledge, the only sequential algorithm in the literature for finding the separations pairs and triconnected components. Indeed the original version of this algorithm has a number of mistakes which have been identified and corrected by Gutwenger and Mutzel [9]. The algorithm divides the separation pairs into two types, Type-1 and Type-2, and needs no less than six passes over the graph for determining them. The strategy involves a *split* operation that divides the input graph into different *split components*: triple bonds, triangles, and triconnected graphs. The input graph G is first split into a set of triple bonds and a graph G'. It then finds all the biconnected components of G'. On each biconnected component the algorithm performs a depth-first search and, based on the results, it runs radix sort with $2|V|+1$ buckets to construct a new adjacency structure, called an *acceptable adjacency structure*, for the graph. It performs a depth-first search again based on the new structure to generate a set of paths. The algorithm runs recursively and each recursive call generates a cycle in the piece of the graph to be tested for separation pairs. This cycle consists of a simple path of edges not in previously found cycles plus a simple path of edges in old cycles. Actually two searches are required for finding the paths. The paths are examined for separation pairs and split components by maintaining the information on two stacks. This is followed by identifying the set of triple bonds and the set of triangles. For each of these two sets the algorithm constructs an auxiliary graph whose vertices are the elements of the set. Eventually, the triple bonds and the triangles are combined into bonds and polygons by finding connected components of corresponding auxiliary graphs.

The triconnectivity algorithm of Hopcroft and Tarjan [11] lacks the simplicity and elegancy of 2-vertex-connectivity and 2-edge-connectivity algorithms [6, 24]. The algorithm is relatively hard to understand. Although it runs with the optimal time and space, the number of passes over the graph and the implementation overhead make the algorithm complicated. In contrast, the recently developed linear time algorithms for 3-edge-connectivity are much simpler in the sense of ease of implementation, number of passes over the graph, and computations involved [16, 23, 26, 27]. The first linear time algorithm for 3-edge-connectivity that was given by Galil and Italiano [7] used the triconnectivity solution of Hopcroft and Tarjan [11] after a linear time reduction from edge-connectivity to vertex-connectivity. Later Taoka et al. [23], Nagamochi et al. [16], and Tsin [26, 27] solved the 3-edge-connectivity problem - with increasing efficiency and simplicity - without using any reduction.

The 3-edge-connectivity solution proposed by Taoka et al. [23] is based on depth-first search. It classifies the cut pairs into two types: *Type-1* which is a pair of a tree edge and a non-tree edge; and *Type-2* which is a pair of two tree edges of a depth-first search tree. First, all Type-1 cut pairs are determined. This is followed by partitioning the graph into disjoint paths so that the edges in every Type-2 cut pair lie on the same path. Once all Type-2 cut pairs are determined, the 3-edge-connected components are computed by adding some new edges to the input graph. The taxonomy of cut pairs by Nagamochi et al. [16] is similar to that by Taoka et al. [23]. Nagamochi et al. [16] performs a depth-first search on the input graph and then counts, for each tree edge on the depth-first search tree, the number of non-tree edges bypassing that tree edge. If a tree edge is bypassed by only one

non-tree edge, then the pair of this tree edge and the bypassing non-tree edge is determined as a Type-1 cut pair. Once all Type-1 cut pairs are determined, the given graph is transformed into a smaller graph. The method is then applied recursively to every non-trivial connected component of the latter which eventually converts all Type-2 cut pairs into Type-1. Both Taoka et al. [23] and Nagamochi et al. [16] require more than one pass of depth-first search. The overall time and space complexity of both of them is $O(|V| + |E|)$.

Some more impressive solutions to the 3-edge-connectivity problem have been given by Tsin [26, 27] based on depth-first search. These algorithms do not even distinguish between Type-1 and Type-2 cut pairs and run in $O(|V| + |E|)$ time and space. The algorithm in [26] can determine all 3-edge-connected components using only one pass over the input graph. The strategy involves an operation, called *absorb-eject*, to transform the input graph into an edgeless graph. Every node of the edgeless graph corresponds to a 3-edge-connected component of the given graph. The most recent 3-edge-connectivity algorithm due to Tsin [27] can determine all cut pairs using only one pass over the input graph. This algorithm [27] does not use any transformation on the input graph and outperforms all the previously known 3-edge-connectivity algorithms in determining cut pairs. The approach is conceptually simpler and exhibits the simplicity and elegancy of existing depth-first search based 2-vertex-connectivity algorithms. Table 1 summarizes the algorithms for 3-vertex-connectivity and 3-edge-connectivity in chronological order.

Problem	**Algorithm due to**	**Year**	**Depends on**	**Number of passes over graph**	**Time & Space Complexity**
3-vertex-conn.	Hopcroft and Tar. [11]	1973	None	≥ 6	$O(\|V\| + \|E\|)$
3-edge-conn.	Galil and Italiano [7]	1991	Hopcroft and Tar. [11]	≥ 7	$O(\|V\| + \|E\|)$
3-edge-conn.	Taoka et al. [23]	1992	None	4	$O(\|V\| + \|E\|)$
3-edge-conn.	Nagamochi et al. [16]	1992	None	≥ 2	$O(\|V\| + \|E\|)$
3-edge-conn.	Tsin [26]	2007	None	1	$O(\|V\| + \|E\|)$
3-edge-conn.	Tsin [27]	2008	None	1	$O(\|V\| + \|E\|)$
3-vertex-conn.	This paper	2008	Tsin [27]	2	$O(\|V\| + \|E\|)$

Table 1. 3-vertex-connectivity and 3-edge-connectivity algorithms

In this paper, we propose a simple 3-vertex-connectivity algorithm of a multigraph by taking the advantage of the 3-edge-connectivity algorithm due to Tsin [27]. We propose a linear time reduction from 3-vertex-connectivity to 3-edge-connectivity of a multigraph. This reduction was previously unknown, while the reduction in the opposite direction already exists [7]. The 3-edge-connectivity algorithm of Tsin [27] is applied on the reduced graph, thereby providing a solution for the 3-vertex-connectivity of the original graph. Hence, for a graph with $|V|$ vertices and $|E|$ edges, the proposed reduction turns into an $O(|V| + |E|)$ time and space algorithm for 3-vertex-connectivity. The algorithm first determines all separations pairs that are necessary and sufficient for determining triconnected components. Upon detection of these separation pairs, finding the triconnected components is trivial. In this way, our approach enjoys the simplicity of the 3-edge-connectivity algorithm. Hence, unlike Hopcroft and Tarjan [11], our algorithm is simple and easy to understand and implement. Furthermore, only a few new terminologies are used. The proposed algorithm, in fact, is an alternative solution to Hopcroft and Tarjan [11], both having optimal time and space.

2 Related Definitions and Notations

A ***connected undirected graph*** is denoted by $G = (V, E)$, where V is the *set of nodes* (or *vertices*) and E is the *set of edges* (or *links*) of G. The number of edges incident on a node v is called the ***degree*** of v and is denoted by $\boldsymbol{deg}(v)$. A depth-first search over an undirected connected graph G generates a spanning tree of G called a ***depth-first search tree***. It labels every edge either as a ***tree edge*** or as a ***non-tree edge***. The search also assigns a distinct number to each node v, called

depth-first search number of v, denoted by ***dfs(v)***, which is the *order* in which the node is visited first time during the search. The *root* of the tree is denoted by r. The terms *path, parent, child, ancestor, descendant* with respect to a depth-first search tree are very common in graph theory and their definitions can be found in [10, 28].

In a depth-first search tree of G, denoted by T, the ***set of children*** of a node $v \in V$ is denoted by ***C(v)***. If $C(v) = \emptyset$, then v is a ***leaf node***. Otherwise, v is a ***non-leaf node***. A pair of nodes is called a ***root leaf pair*** if one node is the root and the other is a leaf node. The ***set of ancestors*** and the ***set of descendants*** of node v are denoted by ***A(v)*** and ***D(v)***, respectively. The sets $A(v) - \{v\}$ and $D(v) - \{v\}$ are called the ***set of proper ancestors*** of v and the ***set of proper descendants*** of v, respectively.

In T, a ***subtree*** rooted at a node u, denoted by $T(u)$, is the subgraph of T induced by $D(u)$. The ***total number of descendants*** of $u \in V$ in T is denoted by n_u. An ***outgoing non-tree edge*** of node v connects v to one of its proper ancestors. An ***incoming non-tree edge*** of v connects v to one of its proper descendants. ***Out(v)*** and ***In(v)*** represent the ***set of outgoing non-tree edges*** of v and the ***set of incoming non-tree edges*** of v, respectively. For a tree edge (u, v), we shall assume that u is the parent of v, while for a non-tree edge (s, t), we shall assume that t is an ancestor of s in the tree, throughout the paper, unless otherwise stated. A tree edge (u, v) is called the ***parent link*** of v and a ***child link*** of u. The parent of node v is denoted by $\boldsymbol{p}(y)$. A $v - y$ ***tree-path*** is the path in T connecting node v to y. Tree edges will be shown by solid lines and non-tree edges will be shown by dotted lines in the figures of the subsequent sections.

$G - X$ represents the graph after removing X from graph G, where X can be a subset of V or a subset of E or a subgraph of G. In G, a ***separation*** (***cut***, respectively) is a minimal non-empty set of nodes (edges, respectively) $F \subseteq V$ ($F \subseteq E$, respectively) such that the total number of components in $G - F$ is greater than that in G. If $|F| = k$, that is, the number of nodes (edges, respectively) in F is k, then F is called a ***k-separation*** (***k-cut***, respectively). The only node (edge, respectively) in a ***1-separation*** (***1-cut***, respectively) is called an ***articulation point*** (a ***bridge***, respectively). A graph G is ***biconnected*** (***bridge-connected***, respectively) if there is no articulation point (bridge, respectively) in G. A *separation* (*cut*, respectively) with two nodes (edges, respectively) is called a ***separation pair*** (***cut pair***, respectively). A detailed characterization of separation pairs is given in Section 3. A graph G is ***k-vertex-connected*** (***k-edge-connected***, respectively) if every separation (cut, respectively) of G has at least k nodes (edges, respectively). A ***k-vertex-connected component*** or a ***k-edge-connected component*** of G is similarly defined. We use the terms *triconnected* and *3-vertex-connected* interchangeably. A more detailed description of these terms is available in [11]. A few more notations will be introduced in the order they appear in the subsequent sections.

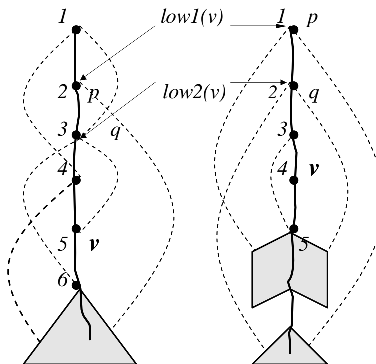


Fig. 1. An illustration of $low1(v)$ and $low2(v)$ (the number beside each node u shows $dfs(u)$)

3 Characterization of Separation Pairs

Let T be a depth-first search tree of an undirected biconnected graph $G = (V, E)$. In T, for each node $v \in V$, we define two terms: $low1(v)$ and $low2(v)$ that were introduced by Hopcroft and Tarjan [11]. For any node v, if $S(v)$ is the set of the depth-first search numbers of v and all t where (s, t) is a non-tree edge and s is a descendant of v and t is a proper ancestor of v, then $low1(v)$ is the smallest number in the set $S(v)$ (Figure 1). The second smallest number in set $S(v)$ will be called $low2(v)$ (Figure 1). These values can be recursively defined as follows.

Definition 1. $low1(v) = \boldsymbol{min}\big(\{dfs(v)\} \cup \{low1(x) | x \in C(v)\} \cup \{dfs(s) | (v, s) \in Out(v)\}\big)$;

Definition 2. $low2(v) = \boldsymbol{min}\big(\{dfs(v)\} \cup \big((\{low1(x) | x \in C(v)\} \cup \{low2(x) | x \in C(v)\} \cup \{dfs(s) | (v, s) \in Out(v)\}) - \{low1(v)\}\big)\big)$;

Let $x, y \in V$ be any pair of nodes such that x is a proper ancestor of y in T. If y is a non-leaf node, then for every $v \in C(y)$, we use $\Delta^{v}_{x,y}$ to denote a ***triple of subgraphs*** (T_1, T_2, T_3) defined as follows (Figure 2):

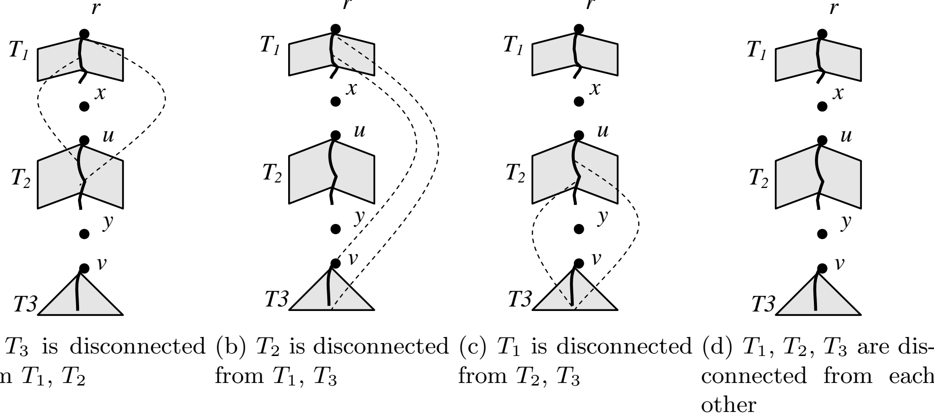
$T_3 = T(v)$;

$T_2 = T(u) - T_3 - \{y\} - T(w)_{all}$;

$T_1 = T - T_2 - T_3 - \{x, y\}$;

where $u \in C(x) \cap A(y)$ and $T(w)_{all}$ represents all the subtrees $T(w)$ such that $w \in C(y)$ and $w \neq v$ and $low1(w) < dfs(x)$;

If y is a leaf node, then (T_1, T_2, T_3) is simply denoted by $\Delta_{x,y}$ and T_3 is a *null graph* with 0 nodes. Let the sets of vertices in subgraphs T_1, T_2, and T_3 be denoted by V_1, V_2, and V_3, respectively.



(a) T_3 is disconnected from T_1, T_2 (b) T_2 is disconnected from T_1, T_3 (c) T_1 is disconnected from T_2, T_3 (d) T_1, T_2, T_3 are disconnected from each other

Fig. 2. An illustration of Theorem 1

Theorem 1. *Let T be a depth-first search tree of an undirected biconnected graph $G = (V, E)$ and Λ be the set of all separation pairs that are necessary and sufficient for determining the triconnected components of G. A pair of nodes $\{x, y\}$ such that $x \in A(y) - \{y\}$ is in Λ if and only if the following is true:*

$\Delta^{v}_{x,y}$ *(if $\exists v \in C(y)$) or $\Delta_{x,y}$ (if y has no child) is (T_1, T_2, T_3) and*

(i) $V_3 \neq \emptyset$, $V_1 \cup V_2 \neq \emptyset$ and there is no non-tree edge (s, t) such that $s \in V_3$ and $t \in V_1 \cup V_2$ or

(ii) $V_2 \neq \emptyset$, $V_1 \cup V_3 \neq \emptyset$ and there is no non-tree edge (s, t) such that $s \in V_2$ ($s \in V_3$, respectively) and $t \in V_1$ ($t \in V_2$, respectively) or

(iii) $V_1 \neq \emptyset$, $V_2 \cup V_3 \neq \emptyset$ and there is no non-tree edge (s, t) such that $s \in V_2 \cup V_3$ and $t \in V_1$.

Proof. If statement (i) is true, then definitely T_3 is disconnected from T_1 and T_2 in $G - \{x, y\}$ (Figure 2(a)). If statement (ii) is true then T_2 is disconnected from T_1 and T_3 (Figure 2(b)) in $G - \{x, y\}$. If statement (iii) is true then T_1 is disconnected from T_2 and T_3 (Figure 2(c)) in $G - \{x, y\}$. If any two of the three statements are true, then all T_1, T_2, and T_3 are disconnected from each other (Figure 2(d)) in $G - \{x, y\}$. Hence, $\{x, y\}$ is a separation pair of G.

To prove in the opposite direction, we first examine which parts of graph G are unaffected by the removal of x and y from G. We can classify all possible connected components of $G - \{x, y\}$ into following 3 groups:

(a) If y is a non-leaf node, then there must be a node $v \in C(y)$ such that the part of G consisting of $T(v)$ and the nodes connected to some node of $T(v)$ by some non-tree edge is unaffected by the removal of x and y and hence remains as a connected component of $G - \{x, y\}$. Let us name this part of the graph as L. L consists of one connected component. If y is a leaf node, then L is a null graph with 0 nodes.

(b) Let $u \in C(x) \cap A(y)$. Then the part of G consisting of the nodes on $u - p(y)$ tree path and all nodes connected to some node on this path by some tree edge or non-tree edge is also unaffected in $G - \{x, y\}$ and remains as a connected component of $G - \{x, y\}$. Let this part of graph be named as M_1. If y has another child $v' \neq v$ such that the outgoing non-tree edges from the nodes of $T(v')$ are incident only to x or y, then $T(v')$ is also unaffected in $G - \{x, y\}$ and remains as a connected component of $G - \{x, y\}$. Let M_2 denotes the subgraph consisting of all such $T(v')$. We call the subgraph consisting of M_1 and M_2 as M. If M_2 does not exist, then M has only one connected component. Otherwise, M consists of more than one connected component.

(c) Let r be the root of T. The part of the graph consisting of nodes on $r - p(x)$ tree path and all nodes connected to the nodes on this path by some tree-edge or non-tree edge are unaffected. We call this part as U. U consists of one connected component.

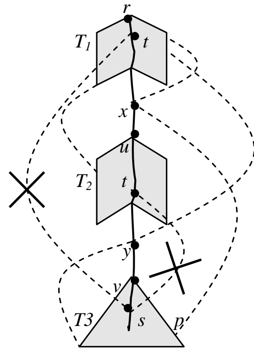
If y is a leaf node then the triple (U, M, L) is unique for $\{x, y\}$, while $\{x, y\}$ will have a triple (U, M, L) for every child of y when y is a non-leaf node. Since $\{x, y\}$ is a separation pair, there must be a triple (U, M, L) for $\{x, y\}$ for which the graph consisitng of L, M, U is disconnected. In fact, combining L, M, and U, we get the graph $G - \{x, y\}$. $G - \{x, y\}$ can be disconnected in two cases: (1) At least one of L, M, U must be different and disconnected from remaining two; (2) M has more than one component. For case (1), at least one of (T_1, T_2, T_3) must be disconnected from the remaining part of $G - \{x, y\}$ since T_1, T_2, and T_3 are subgraphs of U, M, and L, respectively, and, hence, the theorem follows (Figure 2). For case (2), every v' specified in (b) above can be used to define $\Delta_{x,y}^{v'}$ and then, according to (a), $T(v')$ will be L. This $T(v')$ (i.e. T_3 defined by $\Delta_{x,y}^{v'}$) must be disconnected from the remaining part of $G - \{x, y\}$. Thus the scenario reduces to case (1). □

The separation pairs due to statement (i) of Theorem 1 are called ***Type-A*** separation pairs and those due to statement (ii) or statement (iii) are of ***Type-B***. The sets of *Type-A* and *Type-B* separation pairs may not be disjoint. Note that this classification of separation pairs is different from that (Type-1 and Type-2) done by Hopcroft and Tarjan [11].

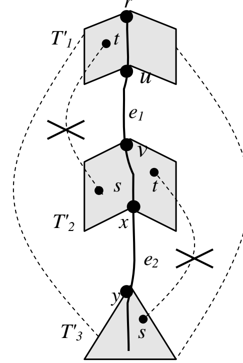
4 The Algorithm

The proposed algorithm for triconnectivity uses a depth-first search on the input graph $G = (V, E)$ resulting into a depth-first search tree T. Any graph having articulation points is first divided into biconnected components and then our algorithm is applied on each biconnected component to find the separation pairs. However, for simplicity, we assume that G is biconnected. The algorithm first determines all separations pairs that are necessary and sufficient for determining triconnected components. Upon detection of these separation pairs, finding the triconnected components is trivial.

During the depth-first search, the algorithm determines *Type-A* separation pairs and performs a reduction on the input graph. *Type-B* separation pairs are determined by applying an existing 3-edge-connectivity algorithm on the reduced graph.



(a) An illustration of Lemma 1



(b) An illustration of Theorem 2 [25]

Fig. 3.

Lemma 1. *Let T be a depth-first search tree of an undirected biconnected graph $G = (V, E)$. Let $x, y \in V$ be a pair of nodes where $x \in A(y) - \{y\}$ in T and $\exists v \in C(y)$ such that $\Delta^v_{x,y} = (T_1, T_2, T_3)$ with $V_3 \neq \emptyset$, $V_1 \cup V_2 \neq \emptyset$. Then $\{x, y\}$ is a Type-A separation pair of G if and only if $low1(v) = dfs(x)$ and $low2(v) \geq dfs(y)$.*

Proof. Let $low1(v) = dfs(x)$ and $low2(v) \geq dfs(y)$. It suffices to show that there is no non-tree edge (s, t) such that $s \in V_3$ and $t \in V_1 \cup V_2$. Let us assume that there is a non-tree edge (s, t) such that $s \in V_3$ and $t \in V_1$ or $t \in V_2$. If $t \in V_1$, then $dfs(t) < dfs(x)$ which contradicts that $low1(v) = dfs(x)$. If $t \in V_2$, then $low2(v) < dfs(y)$ which contradicts that $low2(v) \geq dfs(y)$. Therefore, the non-tree edge (s, t) cannot exist (Figure 3(a)).

On the other hand, if $\{x, y\}$ is a Type-A separation pair, then there is no non-tree edge (s, t) such that $s \in V_3$ and $t \in V_1 \cup V_2$. But there must be some non-tree edge (p, x) such that $p \in V_3$ since G is biconnected. Thus $low1(v) = dfs(x)$ since there cannot be a non-tree edge (s, t) such that s is a descendant of v and t is a proper ancestor of x. And $low2(v) \geq dfs(y)$ because $t \notin V_2$ (Figure 3(a)). □

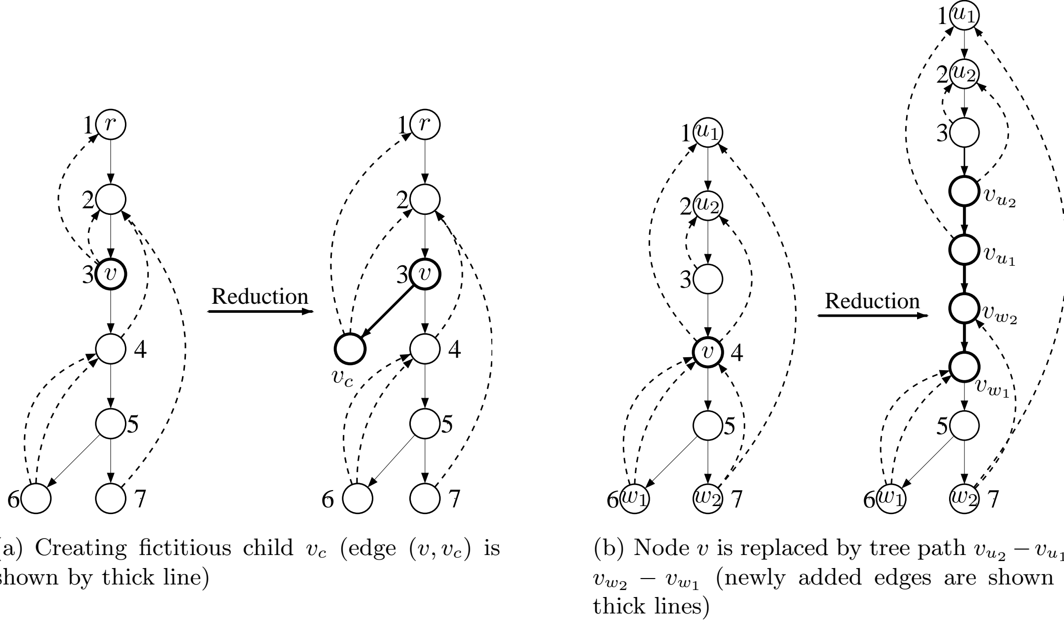
According to Lemma 1, all *Type-A* separation pairs can be detected while performing depth-first search over the graph. *Type-B* separation pairs are determined by reducing this problem to the problem of finding the pairs of tree edges that form cut pairs. Theorem 2 due to Tsin [25] specifies the necessary and sufficient conditions for a pair of tree edges to be a cut pair (Figure 3(b)).

Theorem 2. *In a depth-first search tree of an undirected connected graph $G = (V, E)$, let two tree edges e_1 and e_2 be such that $e_1 = (u, v)$ and $e_2 = (x, y)$, where v is an ancestor of x. Then $\{e_1, e_2\}$ is a cut pair in G if and only if there does not exist a non-tree edge (s, t) such that either $s \in D(v)$ but $s \notin D(y)$ and $t \in A(u)$, or $s \in D(y)$ and $t \in A(x)$ but $t \notin A(u)$.*

Corollary 1 follows from Theorem 1 and Theorem 2.

Corollary 1. *Let T be a depth-first search tree of a biconnected graph G and two tree edges $\{e_1, e_2\}$ be a cut pair of G. Let $\Delta^v_{x,y}$ (if $\exists v \in C(y)$) or $\Delta_{x,y}$ (if y has no child) be (T_1, T_2, T_3) where $x \in A(y) - \{y\}$ and $V_2 \neq \emptyset$ and $\{x, y\}$ is not a root leaf pair. If x is an end node of e_1 and y is an end node of e_2, then $\{x, y\}$ is a separation pair in G.*

For determining *Type-B* separation pairs, the reduction creates a depth-first search tree T' from T. Let the resulting reduced graph of $G = (V, E)$ be denoted by $G' = (V', E')$. That is, T' is a depth-first search tree of G' while T being a depth-first search tree of G. *Type-B* separation pairs



(a) Creating fictitious child v_c (edge (v, v_c) is shown by thick line)

(b) Node v is replaced by tree path $v_{u_2} - v_{u_1} - v_{w_2} - v_{w_1}$ (newly added edges are shown by thick lines)

Fig. 4.

of G are detected after determining the cut pairs in G'. Detection of *Type-A* separation pairs and reduction from G to G' are done by performing a single pass of depth-first search on G.

The values of $low_1(v), low2(v)$, and n_v (number of descendants of v in T), for any node $v \in V$, can be computed in constant time and space while running depth-first search over G. This simple mechanism is similar to Procedure 1 of Hopcroft and Tarjan [11]. When the search backtracks from a node, we check for *Type-A* separation pairs and perform reduction for that node. The entire process is illustrated as TRICONNECTIVITY Algorithm (Algorithm 1).

```
Input: A biconnected graph G = (V, E) represented by adjacency lists Adj(u), ∀u ∈ V;
Output: Type-A separation pairs of G and a reduced depth-first search tree T';
begin
    time := 0;                                         /* initialize depth-first order */
    for each u ∈ V do
        dfs(u) := 0;                                           /* initialize dfs(u) */
        flag(u) := true;    /* indicate that u has not yet been reached by the search */
    end
    Traverse(r, ⊥);       /* search starts at node r, and ⊥ means r will have no parent */
end
```

Algorithm 1: TRICONNECTIVITY

In the algorithm, Procedure *Traverse* mimics depth-first search and examines for *Type-A* separation pairs based on Lemma 1 during the search. For every non-leaf node v, if there exists no $u \in C(v)$ such that $low1(u) = low1(v)$ i.e. $low1(v)$ is defined only by some outgoing non-tree edge of v, then we create a fictitious child v_c of v. The fictitious child v_c becomes a leaf node and all outgoing non-tree edges of v are now outgoing non-tree edges of v_c and, after this, there remains no outgoing non-tree edge of v (Figure 4(a)). The values of $low1(v_c)$ and $dfs(v_c)$ are copied from v. The fictitious children are created so that all *Type-B* separation pairs due to statement (iii) of Theorem 1 (similar to Figure 2(c)) can be treated like those due to statement (ii) of Theorem 1 (similar to Figure 2(b)). For every node v (including the fictitious nodes), if v has at least two non-tree edges incident to it,

```
Traverse(v, y):
begin
    time := time + 1; parent(v) := y; dfs(v) := time;                         /* set dfs(v) */
    low1(v) := low2(v) := dfs(v); n_v := 1;                /* initialize low1, low2, and n_v */
    for each w ∈ Adj(v) do
        if (dfs(w) = 0) then                                               /* explore (v,w) */
            label (v,w) as tree edge;
            Traverse(w, v);
            if (low1(w) < low1(v)) then                           /* update low1(v), low2(v) */
                low2(v) := min(low1(v), low2(w)); low1(v) := low1(w);
            end
            else if (low1(w) = low1(v)) then low2(v) := min(low2(v), low2(w));
            else
                low2(v) := min(low2(v), low1(w));
            end
            n_v := n_v + n_w; parent(w) := v;
        end
        else if ((dfs(w) < dfs(v)) ∧ (w ≠ y ∨ flag(v) = false)) then
            label (v,w) as non-tree edge;
            if (dfs(w) < low1(v)) then                            /* update low1(v), low2(v) */
                low2(v) := low1(v); low1(v) := dfs(w);
            end
            else if (dfs(w) > low1(v)) then  low2(v) := min(low2(v), dfs(w));
        end
        if (w = y) then  flag(v) := false;               /* tree edge (y,v) is examined */
    end
    Let x be the node such that low1(v) = dfs(x);
    if (x ≠ y ∧ low2(v) ≥ dfs(y)) then
         mark {x,y} as a Type-A separation pair ;                              /* by Lemma 1 */
    if ((n_v > 1) ∧ ( there exists no q ∈ C(v) such that low1(v) = low1(q))) then
        Create fictitious child v_c of v;
        Out(v_c) := Out(v); Out(v) := ∅; dfs(v_c) := dfs(v); low1(v_c) := low1(v);
        Reduce(v_c) ;                                           /* perform reduction for v_c */
    end
    if (|Out(v)| + |In(v)| > 1) then       /* at least 2 non-tree edges are incident to v */
        Reduce(v) ;                                               /* perform reduction for v */
    end
end
```

Procedure `Traverse`$(v,\ y)$

Procedure *Traverse* calls subroutine *Reduce*(v) that performs the following reduction to compute final T':

Let $u_1, u_2, \cdots, u_k$ be the sequence of k ($k \geq 0$) nodes connected to v by some outgoing non-tree edges of v such that $dfs(u_i) < dfs(u_{i+1})$, $1 \leq i < k$. Similarly, $w_1, w_2, \cdots, w_l$ is the sequence of l ($l \geq 0$) nodes connected to v by some incoming non-tree edges of v such that $dfs(w_i) < dfs(w_{i+1})$, $1 \leq i < l$. Note that these orderings of neighbors are based on depth-first search order and can be achieved while labelling the non-tree edges during the search. When the search backtracks from a node, the relative orderings of the neighbors are available at that node. As a result, node v can have the above sorted lists of non-tree edges in $O(deg(v))$ time by using a simple counting sort. Then node v is replaced by a tree path $v_{u_k} - v_{u_{k-1}} - \cdots - v_{u_2} - v_{u_1} - v_{w_l} - v_{w_{l-1}} - \cdots - v_{w_2} - v_{w_1}$ such that the non-tree edges between u_i, $1 \leq i \leq k$, (w_i, $1 \leq i \leq l$, respectively) and v are now non-tree edges between u_i (w_i, respectively) and v_{u_i} (v_{w_i}, respectively) and the parent of v is now the parent of v_{u_k} while the children of v being the children of v_{w_1} (Figure 4(b)). If v has only outgoing non-tree edges then it is reduced only to $v_{u_k} - \cdots - v_{u_2} - v_{u_1}$ and the children of v are now children of v_{u_1} (Figure 5(a)). Similarly, if it has only incoming non-tree edges then it is reduced only to $v_{w_l} - v_{w_{l-1}} - \cdots - v_{w_1}$ and the parent of v is now the parent of v_{w_l} (Figure 5(b)). For every

```
Reduce(v):
begin
    Let u_1, u_2, ··· u_k (k ≥ 0) be the sequence of nodes such that ∃(v, u_i) ∈ Out(v) and
    dfs(u_i) < dfs(u_{i+1}), 1 ≤ i < k;

    Let w_1, w_2, ··· w_l (l ≥ 0) be the sequence of nodes such that ∃(w_i, v) ∈ In(v) and
    dfs(w_i) < dfs(w_{i+1}), 1 ≤ i < l;

    Replace v by a tree path v_{u_k} − v_{u_{k−1}} − ··· − v_{u_2} − v_{u_1} − v_{w_l} − v_{w_{l−1}} − ··· − v_{w_2} − v_{w_1};

    parent(v_{u_k}) := parent(v); C(v_{w_1}) := C(v);
    if (k = 0) then parent(v_{w_l}) := parent(v);    /* if v has only incoming non-tree edges  */
    if (l = 0) then C(v_{u_1}) := C(v);              /* if v has only outgoing non-tree edges  */

    for (i := 1 to k) do
        Out(v_{u_i}) := {(v, u_i)|(v, u_i) ∈ Out(v)};     /* non-tree edges (v, u_i) become (v_{u_i}, u_i)  */
        low1(v_{u_i}) := low1(v); dfs(v_{u_i}) := dfs(v);        /* copy low1, dfs for v_{u_i} from v  */
    end

    for (i := 1 to l) do
        In(v_{w_i}) := {(w_i, v)|(w_i, v) ∈ In(v)};      /* non-tree edges (w_i, v) become (w_i, v_{w_i})  */
        low1(v_{w_i}) := low1(v); dfs(v_{w_i}) := dfs(v);        /* copy low1, dfs for v_{w_i} from v */
    end
end
```

Procedure `Reduce(v)`

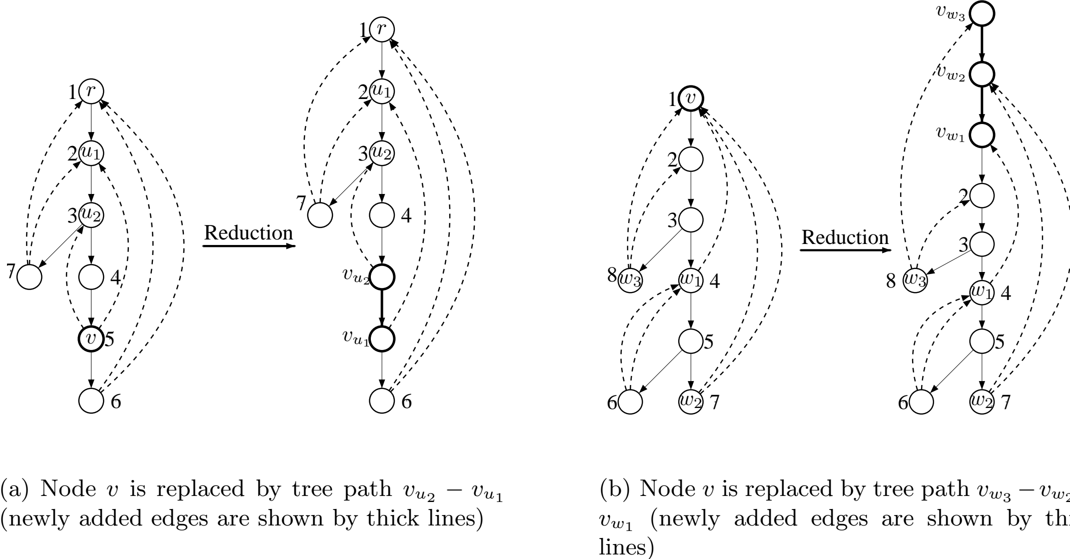
case, the values $low1$ and dfs of every new node are copied from v. If $\{x, y\}$ is a separation pair in G, then, in T', some tree edge either incident to x or generated due to reduction of x will form a separation pair with some tree edge either incident to y or generated due to reduction of y.

For a node $x \in V$ having at least two non-tree edges incident to it in T, let $x_1 - x_2 - \cdots - x_m$ $(m \geq 1)$ be the tree path that replaces x in T', where x_i is the parent of x_{i+1} $(1 \leq i < m)$. In T', P_x denotes a tree path $a - x_1 - x_2 - \cdots - x_m - w$, where a is the parent of x_1, for every child w of x_m in T'. If $x \in V$ has less than two non-tree edges incident to it in T, then P_x, in T', denotes a tree path $a - x - w$, where a is the parent of x, for every child w of x. Note that any node $x \in V$ having less than two non-tree edges and any number of tree edges incident to it in T does not need any reduction because x is already like a node in the reduced graph.

Lemma 2. *Let T be a depth-first search tree of a biconnected graph $G = (V, E)$ and $\Delta^v_{x,y}$ (if $\exists v \in C(y)$) or $\Delta_{x,y}$ (if y has no child) be (T_1, T_2, T_3) where $x \in A(y) - \{y\}$ and $V_2 \neq \emptyset$ and $\{x, y\}$ is not a root leaf pair. Let T' and $G' = (V', E')$ be the reduced graphs of T and G, respectively, from TRICONNECTIVITY Algorithm. Then $\{x, y\}$ is a Type-B separation pair of G if and only if there exist an edge e_1 on some P_x and an edge e_2 on some P_y such that $\{e_1, e_2\}$ is a cut pair of G'.*

Proof. Let $\{x, y\}$ be a *Type-B* separation pair in G. Let, creating all fictitious children in T gives us the depth-first search tree T_c. Let T_c be a depth-first search tree of graph G_c. In $G - \{x, y\}$, if T_1 is disconnected from the rest of the graph, then, in T, $low1(y)$ can be defined by (1) some v' such that $v' \in D(y)$ and v' in T_1 or (2) some outgoing non-tree edge of y, since G is biconnected.

Case (1) refers to statement (iii) of Theorem 1. But considering $\Delta^{v'}_{x,y}$, we will get a different triple of subgraphs (T_1, T_2, T_3) and this case will then refer to statement (ii) of Theorem 1. For Case (2), y must have a fictitious child y_c in T_c if y is a non-leaf node in T. Thus, the triple of subgraphs (T_1, T_2, T_3) of T can be restructured in T_c where y_c will be the root of T_3, and T_2 will be disconnected from T_1 and T_3 in $G_c - \{x, y\}$. Thus, all *Type-B* separation pairs of G due to statement (iii) of Theorem 1 (similar to Figure 2(c)) can be treated like those due to statement (ii) of Theorem 1 (similar to Figure 2(b)) in G_c. That is, any Type-B separation pair in G which is identified using statement (ii) or (iii) of Theorem 1 (like Figure 2(b) or 2(c)) can be identified as a Type-B separation pair in G_c by applying only statement (ii) of Theorem 1 (like Figure 2(b)). Hence, without loss of generality, we can consider the triple of subgraphs (T_1, T_2, T_3) in G_c and call $\{x, y\}$ a Type-B separation pair of G_c which can be identified using only statement (ii) of Theorem 1



(a) Node v is replaced by tree path $v_{u_2} - v_{u_1}$ (newly added edges are shown by thick lines)

(b) Node v is replaced by tree path $v_{w_3} - v_{w_2} - v_{w_1}$ (newly added edges are shown by thick lines)

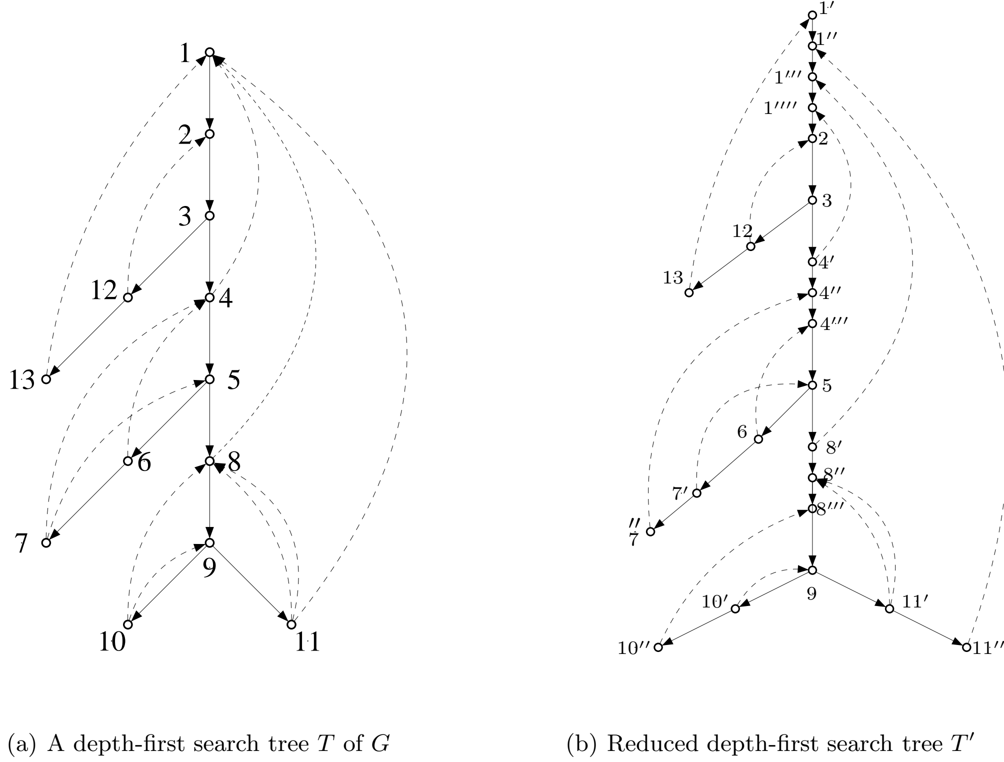
Fig. 5.

(like Figure 2(b)). This implies that T_2 is disconnected from T_1 and T_3 in $G_c - \{x, y\}$. That is, there is no non-tree edge (s, t) such that $s \in V_2$ ($s \in V_3$, respectively) and $t \in V_1$ ($t \in V_2$, respectively). Let the paths P_x and P_y be $x_1 - x_2 - \cdots - x_{k-1} - x_k$ ($k \geq 2$) and $y_1 - y_2 - \cdots - y_{k'-1} - y_{k'}$ ($k' \geq 2$), respectively. Let, the nodes $x_1, x_2, \cdots, x_l$ ($0 \leq l \leq k$) and the nodes $y_{l'}, y_{l'+1}, \cdots, y_{k'}$ ($0 \leq l' \leq k'$) be connected to some ancestor of x_1 or some descendant of y'_k by some non-tree edge. Now, the nodes $x_{l+1}, x_{l+2}, \cdots, x_k$ have incoming non-tree edges only from some non-descendant of of $y_{l'}$. Similarly, the nodes $y_{l'-1}, y_{l'-2}, \cdots, y_1$ have outgoing non-tree edges only to the descendants of x_{l+1}. That is, there is no $z \in V'$ such that z is a descendant of x_{l+1} but not a descendant of $y_{l'}$ and z is connected to an ancestor of x_l or a descendant of $y_{l'}$ by some non-tree edge in T'. By Theorem 2, the edges (x_l, x_{l+1}) and $(y_{l'-1}, y_{l'})$ form a cut pair in G'.

The proof of the opposite direction follows from Corollary 1. □

Once G' is created, we run the 3-edge-connectivity algorithm of Tsin [27] over G' for determining the pairs of tree edges that are cut pairs in G'. The 3-edge-connectivity algorithm performs a single pass of depth-first search. Once the cut pairs in G' are determined, the separation pairs in G can be detected based on Lemma 2. Let $\{e_1, e_2\}$ be a pair of tree edges of T' that has been determined as a cut pair of G' and e_1 and e_2 lie on P_x and P_y, respectively, where $\{x, y\}$ is not a root leaf pair of T. Let $\Delta^v_{x,y}$ be (T_1, T_2, T_3) in T where v is the root of T_3 and u is the root of T_2. For every node $w \in V$, we know n_w (number of descendants of w in T) from Procedure *Traverse*. $|V_3| = n_v$; $|V_2| = n_u - |V_3| - 1 - \sum n_w$, where $w \in C(y) - \{v\}$ and $low1(w) < dfs(x)$; $|V_1| = |V| - |V_2| - |V_3| - 2$. If $|V_2| \neq 0$, then $\{x, y\}$ is a separation pair of G.

Example: Figure 6(a) shows a depth-first search tree T of graph G. The nodes in T are labelled with *dfs* numbers. In the figure, $low1(4) = 1$, $low2(4) = 4$; $low1(5) = 1$, $low2(5) = 4$; $low1(13) = 1$, $low2(13) = 13$; $low1(8) = 1$, $low2(8) = 8$; $low1(6) = 4$, $low2(6) = 5$; $low1(9) = 1$, $low2(9) = 8$; $low1(10) = 8$, $low2(10) = 9$ in T. TRICONNECTIVITY Algorithm determines $\{1, 3\}, \{1, 4\}, \{1, 12\}, \{1, 5\}, \{4, 5\}, \{1, 8\}, \{8, 9\}$ as *Type-A* separation pairs of G and reduces T to T'. The reduced depth-first search tree T' is shown in Figure 6(b). In T', following are the pairs of tree edges that are cut pairs in G': $\{(4', 4''), (5, 8')\}$, $\{(8'', 8'''), (9, 11')\}$, $\{(1', 1''), (12, 13)\}$, $\{(8', 8''), (11', 11'')\}$. By Lemma 2, $\{4, 8\}$, $\{4, 5\}$, $\{8, 11\}$, $\{8, 9\}$, $\{1, 12\}$ are *Type-B* separation pairs of G.



(a) A depth-first search tree T of G

(b) Reduced depth-first search tree T'

Fig. 6.

Theorem 3. *Based on TRICONNECTIVITY Algorithm, all separation pairs that are necessary and sufficient for determining triconnected components of a biconnected graph $G = (V, E)$ with $|V|$ vertices and $|E|$ edges can be determined in $O(|V| + |E|)$ time and space.*

Proof. TRICONNECTIVITY Algorithm has been developed by adding extra instructions in the depth-first search algorithm. Without these added instructions, the basic depth-first search algorithm runs in $O(|V| + |E|)$ time and space. In Procedure *Traverse*, all instructions other than calling Procedure *Reduce* take constant time and space. For a node v that needs to be reduced during the search, Procedure *Reduce* takes $O(deg(v))$ time and space for sorting the neighbors using a simple counting sort and for reduction of v. Considering $V = \{v_1, v_2, \cdots, v_{|V|}\}$, the entire reduction adds $O(deg(v_1) + deg(v_2) + \cdots + deg(v_{|V|})) = O(|E|)$ time and space complexity to the depth-first search. Thus, the overall time and space complexity of TRICONNECTIVITY Algorithm becomes $O(|V|+|E|)$. Since there are $|E|-(|V|-1)$ non-tree edges in a depth-first search tree, the reduction creates a total of $O(|E| - |V| + 1)$ new vertices and edges. Hence, the size of the reduced graph is $O(|V| + |E|)$. The 3-edge-connectivity algorithm of Tsin [27] runs in linear time and space. Hence, all separation pairs of G that are necessary and sufficient for determining triconnected components can be determined in $O(|V| + |E|)$ time and space. □

5 Conclusion

The proposed triconnectivity algorithm makes a total of two passes over the input graph. The first pass performs a depth-first search and detects *Type-A* separation pairs and creates a reduced graph. The second pass involves running the 3-edge-connectivity algorithm of Tsin [27] over the reduced graph which performs a single depth-first search. Finding the cut pairs in the reduced graph gives us the solution of *Type-B* separation pairs of the original graph. Since the algorithm of Tsin [27] solves the 3-edge-connectivity problem while running depth-first search, it might be possible to perform the

computations involved in that algorithm during the first pass of our algorithm which would provide a solution of 3-vertex-connectivity using only one pass over the graph. We leave this as a future work. Our algorithm can determine all the triconnected components by using one more pass over the graph. As mentioned in Section 4, the algorithm, with slight modification, can handle the graphs with articulation points within the same time and space bound. The optimality follows from [11]. Theoretically, both our algorithm and that of Hopcroft and Tarjan [11] have the same optimal time and space complexity. Comparing the performances of these two algorithms in practice is also left as a future work.

References

1. Balakrishnan, R., and Sethuraman, G. Graph theory and its applications. *Alpha Science International* (September 2004).
2. Boffey, T. B. Graph theory in operations research. *Scholium International, Inc* (August 1992).
3. Bondy, J., and Murty, U. Graph theory with applications. *Elsevier Science Ltd* (June 1976).
4. Ellis-Monagham, J. A., and Gutwin, P. Graph theoretical problems in next-generation chip design. *Graphs in Chip Design* (May 1993).
5. Fussell, D. S., Ramachandran, V., and Thurimella, R. Finding triconnected components by local replacements. *SIAM J. Computing 22*, 3 (1993), 587–616.
6. Gabow, H. N. Path-based depth-first search for strong and biconnected components. *Inf. Process. Lett. 74*, 3-4 (2000), 107–114.
7. Galil, Z., and Italiano, G. F. Reducing edge connectivity to vertex connectivity. *SIGACT News 22*, 1 (1991), 57–61.
8. Galil, Z., and Italiano, G. F. Maintaining the 3-edge-connected components of a graph on-line. *SIAM J. Comput. 22*, 1 (1993), 11–28.
9. Gutwenger, C., and Mutzel, P. A linear time implementation of SPQR-trees. In *Graph Drawing, Colonial Williamsburg, 2000* (2001), J. Marks, Ed., Springer, pp. 77–90.
10. Harary, F. Graph theory. *Addison-Wesley* (1969).
11. Hopcroft, J. E., and Tarjan, R. E. Dividing a graph into triconnected components. *SIAM J. Computing 2*, 3 (1973), 135–158.
12. Jungnickel, D. Graphs, networks and algorithms. *Springer (3 edition)* (November 2007).
13. Knauer, B. A simple planarity criterion. *J. ACM 22*, 2 (1975), 226–230.
14. Mathis, P. Graphs & networks. *Wiley-ISTE (2 edition)* (August 2008).
15. Miller, G., and Ramachandran, V. A new graph triconnectivity algorithm and its parallelization. In *STOC '87: Proceedings of the nineteenth annual ACM conference on Theory of computing* (New York, NY, USA, 1987), ACM, pp. 335–344.
16. Nagamochi, H., and Ibaraki, T. A linear time algorithm for computing 3-edge-connected components in a multigraph. *Japan J. Indust. Appl. Math. 8* (1992), 163–180.
17. Nakanishi, N. Graph theory and feynman integrals. *Gordon and Bridge Science Publishers, New York* (1971).
18. Negele, J., and Orland, H. Quantum many-particle systems. *Addison Wesley, Redwood City, CA* (1988).
19. Novak, L., and Gibbons, A. Hybrid graph theory and network analysis. *Cambridge University Press* (September 1999).
20. Poutré, J. A. L. Maintenance of triconnected components of graphs (extended abstract). In *ICALP '92: Proceedings of the 19th International Colloquium on Automata, Languages and Programming* (London, UK, 1992), Springer-Verlag, pp. 354–365.
21. Shaw, P. Private communication. *University of Newcastle, Newcastle, Australia* (2006).
22. sheng Hsu, T., and Ramachandran, V. A linear time algorithm for triconnectivity augmentation (extended abstract). In *IEEE Symposium on Foundations of Computer Science* (1991), pp. 548–559.
23. Taoka, S., Watanabe, T., and Onaga, K. A linear time algorithm for computing all 3-edge-connected components of a multigraph. *IEICE Trans. Fundamentals E75*, 3 (1992), 410–424.
24. Tarjan, R. E. Depth-first search and linear graph algorithms. *SIAM J. Computing 1* (1972), 146–160.
25. Tsin, Y. H. An efficient distributed algorithm for 3-edge-connectivity. *International Journal of Foundations of Computer Science 17*, 3 (2006), 677–701.
26. Tsin, Y. H. A simple 3-edge-connected component algorithm. *Theory of Computing Systems 40*, 2 (February 2007), 125–142.

27. Tsin, Y. H. Yet another optimal algorithm for 3-edge-connectivity. *Journal of Discrete Algorithms* (2008). In press.
28. West, D. B. Introduction to graph theory. *Pearson Education (2 Edition)* (2002), 149.
29. Xu, J. Theory and application of graphs. *Springer (1 edition)* (July 2003).