Self-Stabilizing Computation of 3-Edge-Connected Components

Abusayeed M. Saifullah
Computer Science and Engineering
Washington University
St. Louis, Missouri, USA
Email: saifullaha@cse.wustl.edu

Yung H. Tsin*
School of Computer Science
University of Windsor
Windsor, Ontario, Canada
Email: peter@uwindsor.ca

*Research partially supported by NSERC under grant NSERC-781103.
ABSTRACT

A self-stabilizing algorithm is a distributed algorithm that can start from any initial (legitimate or illegitimate) state and eventually converge to a legitimate state in finite time without being assisted by any external agent. In this paper, we propose a self-stabilizing algorithm for finding the 3-edge-connected components of an asynchronous distributed computer network. The algorithm stabilizes in $O(dn\Delta)$ rounds and every processor requires $O(n \log \Delta)$ bits, where $\Delta(\leq n)$ is an upper bound on the degree of a node, $d(\leq n)$ is the diameter of the network, and $n$ is the total number of nodes in the network. These time and space complexity are at least a factor of $n$ better than those of the previously best-known self-stabilizing algorithm for 3-edge-connectivity. The result of the computation is kept in a distributed fashion by assigning, upon stabilization of the algorithm, a component identifier to each processor which uniquely identifies the 3-edge-connected component to which the processor belongs. Furthermore, the algorithm is designed in such a way that its time complexity is dominated by that of the self-stabilizing depth-first search spanning tree construction in the sense that any improvement made in the latter automatically implies improvement in the time complexity of the algorithm.

KEY WORDS

Distributed system, fault-tolerance, self-stabilization, depth-first search tree, cut-pair, 3-edge-connected component.
1 Introduction

Self-stabilization, first proposed by Dijkstra [6, 7], is a theoretical framework of non-masking fault-tolerance for distributed systems. A self-stabilizing algorithm is a distributed algorithm that can start from any initial (legitimate or illegitimate) state and eventually converge to a legitimate state in finite time without being assisted by any external agent. Thus a self-stabilizing system is capable of tolerating any unexpected transient fault. Many fundamental as well as some advanced graph-theoretic problems in computer network have been studied in the context of self-stabilization over the last decade [1, 2, 3, 5, 10, 11, 12, 13, 14, 15, 20].

The property of edge-connectivity requires considerable attention in graph theory since it measures the extent to which a graph is connected. In telecommunication systems and transportation networks, this property represents the reliability of the network in the presence of link failures. Moreover, when communication links are expensive, it plays a vital role in minimizing the communication cost. Finding \( k \)-edge-connected components, \( k \geq 2 \), is an important issue in distributed computer networks. In a distributed system modelled as an undirected connected graph \( G = (V, E) \), a \( k \)-edge-connected component is defined as a maximal subset \( X \subseteq V \) having the local edge-connectivity at least \( k \) for any \( x, y \in X \), where the local edge-connectivity for two nodes \( x, y \) of \( G \) is the minimum number of edges in \( M \subseteq E \) such that \( x \) and \( y \) are disconnected in \( G - M \), the graph after removing the edge set \( M \) from \( G \).

Several self-stabilizing algorithms for 2-edge-connectivity (as well as 2-vertex-connectivity) are available [3, 5, 12, 13, 14, 20]. Among them, the most efficient one is given by Tsin [20] which stabilizes in \( O(dn\Delta) \) rounds and every processor requires \( O(n \log \Delta) \) bits, where \( \Delta(\leq n) \) is an upper bound on the degree of a node, \( d(\leq n) \) is the diameter of the network, and \( n \) is the total number of nodes in the network. The only known self-stabilizing algorithm for 3-edge-connectivity [17] is a composition of three algorithms that run concurrently where the first algorithm constructs a special spanning tree of the network, called a first depth-first search tree, using the self-stabilizing depth-first search algorithm of Collin et al. [4], the second algorithm computes, for each 3-edge-connected component, all the nodes belong to that component at a specific node belonging to the component and the third algorithm propagates, from each of the specific nodes, the identifiers of all the nodes in the 3-edge-connected components to the remaining nodes in the same component. Hence, the stabilization of the third algorithm depends on the stabilization of the second algorithm while the stabilization of the second algorithm depends on that of the first algorithm. Since the first algorithm
stabilizes in $O(dn^2\Delta)$ steps and each of the other two algorithms stabilizes in $O(nh)$ steps ($h < n$), as was pointed out in Karata et al. [14] and Tsin [20], the worst case time complexity of this composite algorithm is $O(dn^2\Delta \cdot nh \cdot nh) = O(dn^4h^2\Delta)$ steps. If the time complexity is measured in terms of \textit{round}, although the time complexity of the composite algorithm is $O(dn\Delta + h + h) = O(dn\Delta)$, however, each \textit{round} of the composite algorithm will be a \textit{non-constant factor}, namely $n^2h^2$, larger than the \textit{round} of the non-composite algorithm that solves the same problem. The space complexity of the algorithm is $O(n^2\log \Delta)$ bits per processor.

In this paper, we present a new self-stabilizing algorithm, with improved time and space complexity, for 3-edge-connectivity of an asynchronous distributed computer network. The algorithm is non-composite as it runs by itself. The time complexity of the algorithm is $O(dn^2\Delta)$ steps or $O(dn\Delta)$ rounds and every processor requires only $O(n\log \Delta)$ bits. As with the previous algorithm [17], our algorithm also uses a data structure which represents a sequence of ordered pairs of nodes in nested order. However, while the former requires $O(n^2\log \Delta)$ space to maintain the data structure, our algorithm, by using an encoding technique to compress the sequence, needs only $O(n\log \Delta)$ space.

To identify the 3-edge-connected components, we follow the approach used in [3] where a self-stabilizing algorithm is presented to identify 2-edge-connected components. Specifically, when our algorithm stabilizes, each processor is assigned a component identifier to uniquely identify the 3-edge-connected component to which the processor belongs. If every node is to know the set of nodes that are 3-edge-connected to it, then as a node can be 3-edge-connected to $O(n)$ other nodes, just to store the identifiers of these nodes will require $O(n^2\log \Delta)$ space. The space complexity of our algorithm will thus become $O(n^2\log \Delta)$ per processor. However, the time complexity remains unchanged.

Our algorithm is an extension of the algorithm of Collin et al. [4] in the sense that we incorporate the method for computing the 3-edge-connected components into the algorithm so that the components can be determined based on the depth-first search spanning tree. Actually, our method of determining the 3-edge-connected components can be incorporated into any self-stabilizing algorithm that constructs a depth-first search spanning tree. Let $T_{dfs}(n)$ be the time complexity of the self-stabilizing algorithm when it runs on a network of $n$ nodes and $T_3(n)$ be the time complexity of the newly added codes for determining the 3-edge-connected components. Then the time complexity of the resulting self-stabilizing algorithm for 3-edge-connected components is \text{max}\{T_{dfs}(n), T_3(n)\}.

Since $T_{dfs}(n) = O(dn\Delta)$ for the algorithm of Collin et al. [4] and $T_3(n) = O(H\Delta)$, where $H < n$ (to
be shown later), the stated $O(dn\Delta)$ time bound thus follows. Clearly, any improvement made in the time or space complexity for constructing a depth-first search tree would automatically imply an improvement in our algorithm.

2 Some Definitions from Graph Theory

For ease of explanation of the proposed algorithm, some definitions from graph theory are in order. A connected undirected graph is denoted by $G = (V, E)$, where $V$ is the set of nodes and $E$ is the set of edges or links. Two nodes are neighboring if they are connected by an edge and the two nodes are the end-nodes of the edge. In $G$, a non-empty set of edges $M$, $M \subseteq E$, is a cut or an edge-separator if the total number of components in $G - M$ is greater than that in $G$ and no proper subset of $M$ has this property, where $G - M$ represents the graph after removing $M$ from $G$. If $|M| = k$, i.e. the number of edges in $M$ is $k$, then $M$ is called a $k$-cut. The only edge in a 1-cut is called a bridge. A cut with two edges is called a cut-pair or separation-pair. A graph $G$ is $k$-edge-connected if every cut of $G$ has at least $k$ edges. The local edge-connectivity, denoted by $\lambda(x, y; G)$, for two nodes $x$, $y$ of $G$ is the minimum number of edges in $M \subseteq E$ such that $x$ and $y$ are disconnected in $G - M$. A maximal subset $X \subseteq V$ such that $\lambda(x, y; G) \geq k$ for any $x, y \in X$ is called a $k$-edge-connected component of $G$.

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 spanning tree, path, parent, child, ancestor, descendant with respect to a spanning tree are very common in graph theory and their definitions can be found in [9].

In a depth-first search spanning tree of $G$, 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 root-to-leaf path is a path that connects the root $r$ to a leaf node. A $u$-$v$ tree path is a path in the tree connecting nodes $u$ and $v$. The set of ancestors and the set of descendants of node $v$ are denoted by $Anc(v)$ and $Des(v)$, respectively. The sets $Anc(v) - \{v\}$ and $Des(v) - \{v\}$ are called the set of proper ancestors of $v$ and the set of proper descendants of $v$, respectively. A subtree rooted at a node $u$, denoted by $T(u)$, in a tree $T$ is the subgraph of $T$ induced by $Des(u)$. In this paper,
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. A tree edge \((u, v)\) is called the parent link of \(v\) and a child link of \(u\). An outgoing non-tree edge of any node \(v\) connects \(v\) to one of its proper ancestors while an incoming non-tree edge of \(v\) connects \(v\) to one of its proper descendants. \(\text{Out}(v)\) and \(\text{In}(v)\) represent the set of outgoing non-tree edges of \(v\) and the set of incoming non-tree edges of \(v\), respectively.

Lemma 2.1. [16] If nodes \(a\) and \(b\) are 3-edge-connected and nodes \(b\) and \(c\) are 3-edge-connected, then nodes \(a\) and \(c\) are 3-edge-connected.

3 Computational Model

We adopt the model used by Collin and Dolev [4] and Tsin [20]. The distributed system is represented by an undirected connected graph \(G = (V, E)\). The set of nodes \(V\) in \(G\) represents the set of processors \(\{v_1, v_2, \ldots, v_n\}\), where \(n\) is the total number of processors in the system and \(E\) represents the set of bidirectional communication links connecting the processors. We shall use the terms node and processor (edge and link, respectively) interchangeably throughout this paper. There is at most one edge between any two nodes. Without loss of generality, we assume that the graph is bridgeless.

All the processors, except \(v_1\), are anonymous. The processor \(v_1\) is a special processor and is designated as the root. For the processors \(v_i\), \(2 \leq i \leq n\), the subscripts \(2, \ldots, n\) are used for ease of notation only and must not be interpreted as identifiers. Two processors are neighboring if they are connected by a link. The processors run asynchronously and the communication facilities are limited only between the neighboring processors. Communication between the neighbors is carried out using shared communication registers (called registers throughout this paper). Each register is serializable with respect to read and write operations. Every processor \(v_i\), \(1 \leq i \leq n\), contains a register. A processor can both read and write to its own register. It can also read the registers of the neighboring processors but cannot write to those registers. The contents of the registers are divided into fields. Each processor \(v_i\) orders its edges by some arbitrary ordering \(\alpha_i\). For any edge \(e = (v_i, v_j)\), \(\alpha_i(j)\) (\(\alpha_j(i)\), respectively) denotes the edge index of \(e\) according to \(\alpha_i\) (\(\alpha_j\), respectively). Furthermore, for every processor \(v_i\) and any edge \(e = (v_i, v_j)\), \(v_i\) knows the value of \(\alpha_j(i)\).

We consider a processor and its register to be a single entity, thus the state of a processor
fully describes the value stored in its register, program counter, and the local variables. Let $\chi_i$ be the set of possible states of processor $v_i$. A configuration $c \in (\chi_1 \times \chi_2 \times \cdots \chi_n)$ of the system is a vector of states, one for each processor. Execution of the algorithm proceeds in steps (or atomic steps) using read/write atomicity. An atomic step of a processor consists of an internal computation followed by either read or write, but not both. Processor activity is managed by a scheduler (also called daemon). At any given configuration, the scheduler activates a single processor which executes a single atomic step. An execution of the system is an infinite sequence of configurations $\mathcal{R} = (c_0, c_1, \cdots, c_i, c_{i+1}, \cdots)$ such that, for $i \geq 0$, configuration $c_{i+1}$ can be reached from configuration $c_i$ by executing one atomic step. A fair execution is an infinite execution in which every processor executes atomic steps infinitely often. A suffix of a sequence of configurations $(c_0, c_1, \cdots, c_i, c_{i+1}, \cdots)$ is a sequence $(c_k, c_{k+1}, \cdots)$, where $k \geq 0$; the finite sequence $(c_0, c_1, \cdots, c_k)$ is a prefix of the sequence of configurations. A task is defined by a set of executions, called legal executions. A distributed algorithm is self-stabilizing for a task if every fair execution of the algorithm has a suffix belonging to the set of legal executions of that task. The time complexity of the algorithm is expressed in terms of rounds [8]. The first round of an execution $\mathcal{R}$ is the shortest prefix of $\mathcal{R}$ in which every processor executes at least one step. Let $\mathcal{R} = \mathcal{R}_1\mathcal{R}_2$ such that $\mathcal{R}_1$ is the prefix consisting of the first $k$ rounds of $\mathcal{R}$. Then the $(k+1)$-th round of $\mathcal{R}$ is the first round of $\mathcal{R}_2$.

4 Basis of the Algorithm

It is easily verified that if two edges, $e$ and $e'$, form a cut-pair in graph $G$, then at least one of them is a tree edge in a depth-first search spanning tree of $G$. Furthermore, if both $e$ and $e'$ are tree edges, then they lie on a common root-to-leaf path; if one of them, say $e'$, is a non-tree link then link $e$ must lie on the tree-path connecting the two end-nodes of $e'$.

The proposed algorithm is based on depth-first search. In a depth-first search tree, for each node $v \in V$, we compute two terms, $low_1(v)$ and $low_2(v)$, which were introduced in [18] and [19], respectively (Figure 1). (Notations used in Definitions 1 and 4.1 below have been introduced in Section 2.)

Definition 1. $low_1(v) = \min\{dfs(v) \cup \{low_1(x) | x \in C(v)\} \cup \{dfs(s) | (v, s) \in Out(v)\}\}$;
In depth-first search tree, non-tree edges are shown by dotted lines.

Figure 1: An illustration of low1(v) and low2(v) for v ∈ V (In the figure depth-first search starts from the node r).

Definition 2. low2(v) = \[
\begin{cases}
\min\{\{low1(x)\mid x \in C(v) - \{w\}\} \cup \{dfs(s)\mid (v,s) \in Out(v)\} \cup \{dfs(v)\}\}, & \text{if } \exists w \in C(v) \text{ such that } low1(v) = low1(w); \\
\min\{\{low1(x)\mid x \in C(v)\} \cup \{dfs(s)\mid (v,s) \in Out(v) \land dfs(s) \neq low1(v)\} \\
\cup \{dfs(v)\}\}, & \text{otherwise } ;
\end{cases}
\]

Specifically, low1(v) is the dfs number of the vertex, u, that is closest to the root and is reachable from vertex v by traversing zero or more tree edges following by exactly one non-tree edge. Excluding the edges on the aforementioned path connecting v with u, low2(v) is the dfs number of the vertex that is closest to the root and is reachable from vertex v by traversing zero or more tree edges following by exactly one non-tree edge. Figure 1 gives an illustration of these terms.

Since every depth-first search number is unique in a depth-first search tree, for any node v, the notation dfs(v) will often be used to denote the node v for ease of presentation of our algorithm. The following lemma is easily verified.

Lemma 4.1. Every node v is 3-edge-connected to low2(v).

Definition 3. An incident link (v, w) of a node v is a to-low link of v if w ∈ C(v) and low1(w) = low1(v), or (v, w) ∈ Out(v) and dfs(w) = low1(v). In the former case, node w is called a lowchild of v.

Note that to-low link (lowchild, respectively) of a node is non-unique. However, in the algorithm to be presented below, the first incident link from which node v receives the final value of low1(v) is designated as the to-low link of v; the corresponding lowchild, if exists, is designated as the lowchild of v.
Definition 4. The to-low path of node $v$ is the longest path starting from $v$ and consisting of to-low links of descendants of $v$.

Lemma 4.2. The to-low path of $v$ is a $v$–low1($v$) path in which every link is a tree link except the last one.

Proof: By induction on the length of the to-low path. \qed

The correctness of the proposed algorithm is based on the following characterization theorem for cut-pairs which is a generalization of Theorem 1 in [19].

Theorem 4.3. Given a depth-first search tree rooted at $r$, two edges $e = (v, w)$ and $e' = (x, y)$ form a cut-pair if and only if (assuming without loss of generality that $y$ is an ancestor of $v$) (i) for every node $u$ lying on the $y$–$v$ tree path, there does not exist an incoming non-tree link $(s, u)$ such that $s$ is a descendant of $w$, and (ii) if $(x, y)$ is a tree link, then for every node $u$ lying on the $r$–$x$ tree path, there does not exist an incoming non-tree link $(s, u)$ such that $s$ is a descendant of $y$ but not of $w$.

Proof: Similar to the proof of Theorem 1 in [19]. \qed

Corollary 4.3.1. Let $(x, y)$ and $(v, w)$ form a cut-pair such that $(x, y)$ is a tree link and $y$ is an ancestor of $v$. Then $(v, w)$ must lie on the to-low path of $y$.

Corollary 4.3.2. Let nodes $a$, $b$, $c$, and $d$ be such that $a$ is a proper ancestor of $b$, $b$ is an ancestor of $c$, and $c$ is an ancestor of $d$. Suppose $a$ and $c$ are 3-edge-connected and either $b$ and $d$ are 3-edge-connected or $\exists (s, b) \in In(b)$ such that $s$ is a descendant of $d$. Then $b$ and $c$ are 3-edge-connected.

5 Description of the Self-Stabilizing 3-Edge-Connectivity Algorithm

To determine the 3-edge-connected components, we shall identify a unique node in each 3-edge-connected component and use the identity of that node to label all the nodes in that 3-edge-connected component. We shall first give a characterization of those nodes. For ease of explanation, we shall use $a \prec b$ ($a \preceq b$, respectively) to denote ‘node $a$ is a proper ancestor (ancestor, respectively) of node $b$.'
Lemma 5.1. Let $u$ be a node such that the parent link of $u$ does not form a cut-pair with any link in $T(u)$ (the subtree rooted at $u$) or any non-tree link having an end-node in $T(u)$. Then $\text{low}_2(u) \prec u$ or there exists a sequence of nodes $w_i, 1 \leq i \leq k$, on the to-low path of $u$ such that (Figure 2):

(i) $\text{low}_2(w_1) \prec u \prec w_1$;

(ii) $w_1 \preceq w_2$ and there is a non-tree link $(a, b)$ such that $u \preceq b \prec w_1$ while $w_2$ is the closest ancestor of $a$ on the to-low path of $u$;

(iii) $b \preceq w_3 < w_1$ and $\text{low}_2(w_3) < b$;

(iv) $\text{low}_2(w_3) \preceq w_4 < b$ and $\text{low}_2(w_4) < \text{low}_2(w_3)$;

(v) $\text{low}_2(w_{i-1}) \preceq w_i < \text{low}_2(w_{i-2}), 5 \leq i \leq k$, and $\text{low}_2(w_i) < \text{low}_2(w_{i-1}), 5 \leq i \leq k$;

(vi) $\text{low}_2(w_k) = u$.

Proof: See Appendix A.

Theorem 5.2. A node $u$ is an ancestor of all the other nodes in the 3-edge-connected component it belongs to if and only if the parent link of $u$ forms a cut-pair with some link in $T(u)$ or having an end-node in $T(u)$.

Proof: See Appendix B.

Definition 5. The representative node of a 3-edge-connected component is the node in that component that is an ancestor of all the other nodes in that component in a depth-first search spanning
tree. For each node w, the representative node of the 3-edge-connected component containing w is denoted by \text{reprenode}(w).

Owing to Theorem 5.2, every node u whose parent link does not form a cut-pair with some link in T(u) or some link having an end-node in T(u) is a representative node. It remains to find an effective way of determining \text{reprenode}(u) at every node u.

**Definition 6.** A sequence of ordered pairs of nodes, \((x_1, q_1), (x_2, q_2), \ldots, (x_k, q_k)\), is in nested order if \(x_{j+1}\) is an ancestor of \(x_j\); \(q_{j+1}\) is a descendant of \(q_j\), \(1 \leq j < k\), and node \(x_k\) is a descendant of \(q_k\).

**Definition 7.** Two ordered pairs of nodes \((u, v)\) and \((x, y)\) interlace if \(v\) is an ancestor of \(y\), \(y\) is an ancestor of \(u\), and \(u\) is an ancestor of \(x\).

In order to determine the representative node of every 3-edge-connected component, every node \(v\) maintains a sequence of ordered pairs of nodes, \(S_v : (x_1, q_1), (x_2, q_2), \ldots, (x_k, q_k)\), in nested order such that node \(v\) is an ancestor of \(x_k\) and a descendant of \(q_k\). Furthermore, \(x_i, 1 \leq i \leq k\), and \(q_i\) are 3-edge-connected to each other and all nodes \(x_j, 1 \leq j \leq k\), lie on the to-low path of \(v\). The sequence indicates that the parent link of \(q_1\) has the potential of forming a cut-pair with a link on the to-low path of \(x_1\), and the parent link of \(q_i, 2 \leq i \leq k\), has the potential of forming a cut-pair with a link on the \(x_{i-1} - x_i\) tree-path. Therefore, each \(q_i, 1 \leq i \leq k\), is a potential representative node of a 3-edge-connected component.

The sequence \(S_v\) is constructed as follows: node \(v\) reads the sequence of node-pairs, \(S_w : (x_1, q_1), (x_2, q_2), \ldots, (x_k, q_k), k \geq 0\), from its lowchild \(w\); if \(v\) has no lowchild, then \(S_w\) is an empty sequence. Node \(v\) then modifies \(S_w\) so as to produce \(S_v\) as follows:

(i) Node \(v\) calculates \(\text{next}(v)\) which is defined as:

**Definition 8.**

Let \(S_w : (x_1, q_1), (x_2, q_2), \ldots, (x_k, q_k), k \geq 0\).

\[ \text{next}(v) = \min_\prec \left\{ \{\text{low2}(v)\} \cup \{q_j \mid \exists (s, v) \in \text{In}(v) \text{ such that } (x_j, q_j) \text{ interlaces with } (s, v)\} \right\}. \]

Specifically, \(\text{next}(v)\) is either the node \(\text{low2}(v)\) or the node \(q_f\) with the smallest index \(f\) such that \((x_f, q_f)\) interlaces with some incoming non-tree link of \(v\). Node \(v\) then removes every node-pair \((x_i, q_i)\) from \(S_w\) such that \(q_i\) is a proper descendant of \(\text{next}(v)\). This is because, by Theorem 4.3, the parent link of \(q_i\) cannot form a cut-pair with a link lying on the \(x_{i-1} - x_i\) path and hence is no longer a potential representative node of a 3-edge-connected component.
(ii) If $\text{next}(v) = v$, then node $v$ is the representative node of a 3-edge-connected component (see Theorem 5.4 below). Otherwise, if $\text{next}(v) = q_f$, then the node-pair $(x_f, q_f)$ is replaced by $(v, \text{next}(v))$, and if $\text{next}(v) < q_f$, then $(v, \text{next}(v))$ is simply added to $S_w$ as the innermost node-pair. This is because the parent link of $\text{next}(v)$ has the potential of forming a cut-pair with some link lying on the $v - x_{f-1}$ tree-path. In either case, the modified $S_w$ becomes $S_v$.

**Lemma 5.3.** For every node $u$, $\text{next}(u)$ is 3-edge-connected to $u$.

**Proof:** See Appendix C.

**Theorem 5.4.** Node $u$ is a representative node if and only if $\text{next}(u) = u$.

**Proof:** See Appendix D.

### 6 Adoption of Self-Stabilization

Since our algorithm uses the self-stabilizing depth-first search algorithm of Collin and Dolev [4] to construct a depth-first search spanning tree of the given network, to make our presentation self-contained, we shall give a brief overview of their algorithm.

In the self-stabilizing depth-first search algorithm of Collin and Dolev, every processor $v_i$ has a field, denoted by $\text{path}_i$, in its register. At any point of time during the execution of the algorithm, $\text{path}_i$ contains the sequence of indices of the links on a path connecting the root $v_1$ with node $v_i$. The algorithm uses a lexicographical order relation $\prec$ on the path representation. Specifically, $\text{path}_i \prec \text{path}_j$ if and only if $\text{path}_j = \text{path}_i \oplus s$, for some $s$, where $\oplus$ is the concatenation operator. During the execution of the algorithm, the root processor $v_1$ repeatedly writes $\perp$ in its $\text{path}_1$ field and, in the lexicographical order relation, $\perp$ is the minimal element. The remaining processors repeatedly calculate the smallest (with respect to the lexicographical order $\prec$) path connecting $v_1$ with themselves by reading the $\text{path}$ values from the registers of their neighboring processors and store the calculated result in the $\text{path}$ field of their own registers. When the algorithm stabilizes, the last links on the smallest paths of $v_i$, $i \geq 2$, form a depth-first search tree of the network, called the first depth-first search tree.

Since in the first depth-first search tree, a node $v_j$ is an ancestor of a node $v_i$ if $\text{path}_j \prec \text{path}_i$, and $v_j$ is the parent node of $v_i$ if $v_j$ is the unique neighbor of $v_i$ such that $\text{path}_i = \text{path}_j \oplus \alpha_j(i)$, $\text{path}_i$ can play the role of $\text{dfs}(v_i)$. The definitions of $\text{low}_1$ and $\text{low}_2$ can thus be rewritten as follows (where function $\text{min}_{\prec}$ returns the lexicographically minimum path): $\forall v_i$, $1 \leq i \leq n$, 

$$
\text{low}_1(v_i) = \text{min}_{\prec} \{ \text{path}_j : j \prec i \}
$$

$$
\text{low}_2(v_i) = \text{min}_{\prec} \{ \text{path}_j : j \succ i \}
$$
Definition 9. \( \text{low}_1 = \min_{\alpha} \{ \{ \text{path}_i \} \cup \{ \text{low}_1 \mid v_j \in C(v_i) \} \cup \{ \text{path}_j \mid (v_i, v_j) \in \text{Out}(v_i) \} \} \); \( \text{low}_2 = \begin{cases} \min_{\alpha} \{ \{ \text{low}_1 \mid v_j \in C(v_i) \} \cup \{ \text{path}_j \mid (v_i, v_j) \in \text{Out}(v_i) \} \cup \{ \text{path}_i \} \}, & \text{if } \exists v_k \in C(v_i) \text{ such that } \text{low}_1 = \text{low}_1_k; \\ \min_{\alpha} \{ \{ \text{low}_1 \mid v_j \in C(v_i) \} \cup \{ \text{path}_j \mid (v_i, v_j) \in \text{Out}(v_i) \} \}, & \text{otherwise}. \end{cases} \)

The degree of a node \( v_i \), denoted by \( \delta_i \), is the number of incident links on \( v_i \). A string \( s' \) is a prefix of a string \( s \) if \( (\exists s'')(s = s' \oplus s'') \). Once the depth-first search tree is constructed, at each node \( v_i \), the type of each incident link \( (v_i, v_j) \) (or \( (v_j, v_i) \)) can be determined by \( \text{path}_i, \text{path}_j, \alpha_i(j), \) and \( \alpha_j(i) \) as follows:

- The link \( (v_j, v_i) \) is the parent link if and only if \( \text{path}_i = \text{path}_j \oplus \alpha_j(i) \);
- The link \( (v_i, v_j) \) is a child link if and only if \( \text{path}_j = \text{path}_i \oplus \alpha_i(j) \);
- The link \( (v_i, v_j) \) is an outgoing non-tree edge if and only if \( (\exists s)((\text{path}_i = \text{path}_j \oplus s) \land (s \neq \alpha_j(i))) \);
- The link \( (v_j, v_i) \) is an incoming non-tree edge if and only if \( (\exists s)((\text{path}_j = \text{path}_i \oplus s) \land (s \neq \alpha_i(j))) \).

To incorporate our method of determining 3-edge-connected components into the self-stabilizing algorithm of Collin et al. [4], we must explain how to compute the various values such as \( \text{low}_1, \text{low}_2 \), and \( \text{next} \) based on the depth-first search tree, henceforth denoted by \( T_{\text{dfs}} \), constructed by their algorithm.

Along with the \( \text{path}_i \) field, every processor \( v_i, i \geq 2 \), maintains some additional fields: \( \text{low}_1 \) (Definition 9), \( \text{low}_2 \) (Definition 10), \( \text{nestedpath}_i \), \( \text{next}_i \), \( \text{rtcc}_i \), \( \text{tcc}_i \) in its register. The special processor \( v_1 \) (root) maintains only the fields \( \text{rtcc}_1 \) and \( \text{tcc}_1 \) in addition to \( \text{path}_1 \). When the algorithm stabilizes, at every processor \( v_i, 1 \leq i \leq n \), the field \( \text{tcc}_i \) contains the path value of \( \text{reprenode}(v_i) \).

The field \( \text{nestedpath}_i \) is used to represent the node-pair sequence, \( S_{v_i} : (x_1, q_1), (x_2, q_2), \ldots, (x_k, q_k) \), defined in the previous section. However, instead of representing the sequence with a sequence of \( 2k \) \( \text{path} \) values, we shall represent the sequence in a compact form, using only one \( \text{path} \) value. This is possible because \( x_1 \) is a descendant of \( x_i, 2 \leq i \leq k \) and \( q_i, 1 \leq i \leq k \). Therefore, the path values of \( x_i, 1 \leq i \leq k \), and \( q_i, 1 \leq i \leq k \), can all be marked in a path value, \( \text{path} \), such that \( x_1 \preceq \text{path} \). The content of \( \text{nestedpath}_i \) has the following structure:
• For $x_j, 1 \leq j \leq k$, let $path = path_{x_j} \oplus s$. There is a $\$ symbol in between $path_{x_j}$ and $s$ in $path$;

• For $q_j, 1 \leq j \leq k$, let $path = path_{q_j} \oplus s$. There is a $\$ symbol in between $path_{q_j}$ and $s$ in $path$.

Specifically, for the node-pair, $(x_j, q_j)$, in $S_v$, the prefix of $nestedpath_i$ terminated by the $j$-th (from the beginning) $\$ symbol is $path_{q_j}$ while that terminated by the $j$-th (from the end) $\$ symbol is $path_{x_j}$ after the intervening $\$ symbols are removed.

For example, let $nestedpath_i$ be $\alpha_1 \alpha_2 \$ \alpha_3 \$ \alpha_4 \alpha_5 \$ \alpha_6 \alpha_7 \$ \alpha_8 \$ \alpha_9 \alpha_{10} \$, where each $\alpha_j, 1 \leq j \leq 10$, denotes an edge index. Then,

\[
\begin{align*}
path_{x_1} &= \alpha_1 \alpha_2 \alpha_3 \alpha_4 \alpha_5 \alpha_6 \alpha_7 \alpha_8 \alpha_9 \alpha_{10} \text{ (the subsequence of indices up to the last $\$ symbol)}; \\
path_{q_1} &= \alpha_1 \alpha_2 \text{ (the subsequence of indices up to the 1st $\$ symbol)}; \\
path_{x_2} &= \alpha_1 \alpha_2 \alpha_3 \alpha_4 \alpha_5 \alpha_6 \alpha_7 \alpha_8 \text{ (the subsequence of indices up to the 2nd last $\$ symbol)}; \\
path_{q_2} &= \alpha_1 \alpha_2 \alpha_3 \text{ (the subsequence of indices up to the 2nd $\$ symbol)}; \\
path_{x_3} &= \alpha_1 \alpha_2 \alpha_3 \alpha_4 \alpha_5 \alpha_6 \alpha_7 \text{ (the subsequence of indices up to the 3rd last $\$ symbol)}; \\
path_{q_3} &= \alpha_1 \alpha_2 \alpha_3 \alpha_4 \alpha_5 \text{ (the subsequence of indices up to the 3rd $\$ symbol)}. 
\end{align*}
\]

Furthermore, the nested sequence of node-pairs is: $(x_1, q_1), (x_2, q_2), (x_3, q_3)$.

If $next_i = v_i$, then by Theorem 5.4, node $v_i$ is the representative node of the 3-edge-connected component containing it. Node $v_i$ will use $path_i$ as the identifier for the 3-edge-connected component. Since $v_i$ is an ancestor of all the other nodes in that component, the identifier can thus be propagated downward within $T_{dfs}$ as follows: every node $v_i$ keeps the $path$ values of its ancestors that are representative nodes in a compact form in the field $rtcc_i$. The node reads the $rtcc$ field of its parent node and uses $next_i$ to retrieve the $path$ value of $reprenode(v_i)$ and stores it in the field $tcc_j$ in its register. When the algorithm stabilizes, all the nodes of the same 3-edge-connected component contain the same distinct $tcc$ value.

The following subsections describe the computation of different fields in the register of every non-root node $v_i, i \geq 2$. For ease of presentation, we let $v_{ij}, 1 \leq j \leq \delta_i \text{ (the degree of } v_i), \text{ be the neighboring processors of processor } v_i, 1 \leq i \leq n, \text{ such that } \alpha_i(i_j) = j, 1 \leq j \leq \delta_i, 1 \leq i \leq n$. The functions $read$ and $write$ are the functions for reading from and writing to a register, respectively.

### 6.1 Computing $path_i$

Procedure $ComputePath(v_i, v_{ij}, 1 \leq j \leq \delta_i)$ shows how every processor $v_i, i \geq 2$, computes $path_i$. The function $trunc_N$ returns the rightmost $N$ items of its argument, where $N(\geq n)$ is an
upper bound on the number of processors. The details about the computation of \( \text{path}_i \) are available in [4].

\[
\text{ComputePath}(v_i, v_{ij}, 1 \leq j \leq \delta_i) \quad \text{/* Procedure for computing \( \text{path}_i, i \geq 2 */}
\begin{align*}
\text{begin} & \\
\text{for } j := 1 \text{ to } \delta_i \text{ do } & \text{readpath}_j := \text{read}(\text{path}_{ij}); \quad \text{/* read the path value of neighbor } v_{ij} \text{ into local variable readpath}_j */ \\
\text{write } & \quad \text{path}_i := \min \{ \text{trunc}_N(\text{readpath}_j \oplus \alpha_{ij}(i)) | 1 \leq j \leq \delta_i \}; \quad \text{/* compute path}_i */ \\
\text{end}
\end{align*}
\]

Procedure \text{ComputePath}(v_i, v_{ij}, 1 \leq j \leq \delta_i)

Lemma 6.1. For every fair execution of Procedure \text{ComputePath}, given that \( \text{path}_1 = \bot \) in every configuration, where \( v_1 \) is the root of \( T_{dfs} \), there is a suffix \( \Pi \) in which, in every configuration, \( \text{path}_i, 1 \leq i \leq n \), is the smallest path connecting \( v_1 \) and node \( v_i \).

\begin{proof}
Immediate from Theorem 3.3 in [4]. \qed
\end{proof}

6.2 Computing \( \text{low}_1, \text{low}_2 \)

Every processor \( v_i, i \geq 2 \), calls the procedure \text{ComputeLow}(v_i, v_{ij}, 1 \leq j \leq \delta_i) \) for computing \( \text{low}_1 \) and \( \text{low}_2 \). Each leaf node \( v_i \) computes \( \text{low}_1 \) and \( \text{low}_2 \) based on the \( \text{path} \) values it reads from its outgoing non-tree links (lines 11-14). Each non-leaf node \( v_i \) computes \( \text{low}_1 \) and \( \text{low}_2 \) based on the \( \text{low}_1 \) values it reads from its children (lines 6-10) and the \( \text{path} \) values it reads from its outgoing non-tree links (lines 11-14). The procedure also records a \( \text{lowchild} \) (Definition 4) for \( v_i \). If \( \text{low}_1 \) is defined from some outgoing non-tree link of \( v_i \), then \( \text{lowchild}_i \) is recorded as \text{null} meaning that \( v_i \) has no \text{lowchild} (lines 12-13). Note that \( \text{readpath}_j \) was initialized in Procedure \text{ComputePath}.

Lemma 6.2. For every fair execution of Procedure \text{ComputeLow}, if there is a suffix \( \Pi \) in which \( \text{path}_i, 1 \leq i \leq n \), contain the correct values in every configuration, then there is a suffix of \( \Pi \) in which \( \text{path}_i, \text{low}_1, \text{low}_2, \) and \( \text{lowchild}_i, 1 \leq i \leq n \), contain their correct values in every configuration.

\begin{proof}
After the execution reaches the first configuration of \( \Pi \), since \( \text{path}_i, 1 \leq i \leq n \), contain the correct values, therefore at every leaf node \( v_i \) of \( T_{dfs} \), after reading in the \( \text{path} \) values from all the outgoing non-tree links, \( \text{low}_1 \), \( \text{low}_2 \) are correctly computed. Moreover, \( \text{lowchild}_i \) is also correctly set to \text{null} as the to-low link is a non-tree link. Therefore, there is a suffix of \( \Pi \) in which \( \text{path}_i, \text{low}_1, \text{low}_2 \) and \( \text{lowchild}_i \), contain the correct values at every leaf-node \( v_i \) of \( T_{dfs} \). Suppose
ComputeLow($v_i, v_{ij}, 1 \leq j \leq \delta_i$): /* Procedure for computing low1, low2, $i \geq 2$ */
begin
low1 := low2 := path := read(pathi); /* initialize low1, low2, and path */
lowchild := null; /* initialize lowchild */
for $j := 1$ to $\delta_i$ do
if (readpathj = path $\oplus j$) then /* $(v_i, v_{ij})$ is a child link */
readlow1 := read(low1ij); /* read low1 value of child $v_{ij}$ */
if (readlow1 < low1) then /* update low1, lowchild */
low2 := low1; low1 := readlow1; lowchild := readpathj; /* and low2 */
else low2 := min$_{\prec}$ (low2, readlow1j); /* update low2 */
else if ($\exists s ((\text{path} = \text{readpath}_{j} \oplus s) \land (s \neq \alpha_{ij}(i)))$) then /* the link $(v_i, v_{ij})$ is an outgoing non-tree edge */
if (readpathj < low1) then /* update low1, lowchild */
low2 := low1; low1 := readpathj; lowchild := null; /* and low2 */
else low2 := min$_{\prec}$ (low2, readpathj); /* update low2 accordingly */
write low1 := low1; write low2 := low2; write lowchild := lowchild;
end

Procedure ComputeLow($v_i, v_{ij}, 1 \leq j \leq \delta_i$)

there is a suffix $\Pi'$ of $\Pi$ in which in every configuration, low1, low2, $1 \leq j \leq n$, contain the correct values at every node $v_j$ on level $h$ or higher (i.e. farther from $v_1$). For every node $v_i$ on level $h - 1$, based on the correct low1 values of the child nodes (on level $h$) and the correct values of path$_j$, $1 \leq j \leq n$, low1, low2, and lowchild, are correctly determined. Hence, there is a suffix of the execution in which, for every node $v_i$, $1 \leq i \leq n$, path$_i$, low1, low2, and lowchild$_i$, $1 \leq i \leq n$, contain their correct values.

6.3 Computing nestedpath$_i$

For clarity, we shall use $x_j$ and $q_j$ to represent path$_{x_j}$ and path$_{q_j}$, respectively, in the presentation below.

Every processor $v_i$, $i \geq 2$, calls procedure ComputeNestedPath($v_i, v_{ij}, 1 \leq j \leq \delta_i$) for computing nestedpath$_i$ which is a compact representation of the sequence of nested node-pairs $S_{v_i}$.

In the procedure, the function mark(nestedpath, (path$_1$, path$_2$)) places in nestedpath, a $\$$ symbol right after path$_1$ and a $\$$ symbol right after path$_2$ (ignoring the intervening $\$$ symbols), so that the pair (path$_1$, path$_2$) can be retrieved later. The function unmark(nestedpath, (path$_1$, path$_2$)) removes the $\$$ symbol following path$_1$ and the $\$$ symbol following path$_2$ in nestedpath.
1. ComputeNestedPath(v_i, v_i, 1 ≤ j ≤ δ_i): /* Computing nestedpath_i, i ≥ 2 */
2. begin
3.  next := low2_i := read (low2_i); nestedpath := path := read (path_i);
4.  lowchild := read (lowchild_i);
5.  if (lowchild ≠ null) then
6.     Let lowchild = path_i be the lowchild;
7.     nestedpath := read (nestedpath_i);
8.     Let the pairs of path values marked in nestedpath be (x_1, q_1), (x_2, q_2), ..., (x_k, q_k) in
9.     nested order where for any pair (x_i, q_i), 1 ≤ l ≤ k, q_i and x_i are the path values up to
10.    the l-th (from start) $ symbol and up to l-th (from end) $ symbol, respectively, in
11.    nestedpath;
12.    if (readpath_i = q_k) then /* unmark innermost pair */
13.        nestedpath := unmark(nestedpath, (x_k, q_k));
14.        /* Now unmark all pairs interlaced with (v_i, low2_i) */
15.        for every (x_i, q_i), 1 ≤ l ≤ k, marked in nestedpath such that low2_i ≤ q_i ≤ path_i do
16.            nestedpath := unmark(nestedpath, (x_i, q_i));
17.        end
18.    end
19. for j := 1 to δ_i do /* If the link (v_i, v_i) is an incoming non-tree edge, then unmark all
20.     pairs interlaced with (v_i, v_i) */
21.     if (∃s)((readpath_j = path ⊕ s) ∧ (s ≠ α_s(i_j))) then
22.         for every (x_i, q_i), 1 ≤ l ≤ k, in nestedpath such that path ≤ x_i ≤ readpath do
23.             nestedpath := unmark(nestedpath, (x_i, q_i));
24.             next = min_<(q_i, next); /* update next */
25.         end
26.     end
27. /* Insert innermost pair in nestedpath and write in register */
28. write nestedpath_i := mark(nestedpath, (path, next)); write next_i := next;
29. end

Procedure ComputeNestedPath(v_i, v_i, 1 ≤ j ≤ δ_i)

Initially, at any node v_i, the low2_i and path_i values are used to initialize the local variables next and nestedpath, respectively (line 3). If v_i has no lowchild, then no computation takes place in lines 6-20 of the procedure. On line 21, path_i and next_i (which is low2_i) are marked in nestedpath and the resulting value is recorded into the field nestedpath_i in the register. This effectively creates the nested sequence S_{v_i} : (v_i, low2(v_i)) and the next value is also copied into the field next_i.

If v_i has a lowchild v_{ic}, then nestedpath_i is read into the local variable nestedpath (line 7). Let the sequence in nestedpath be (x_1, q_1), (x_2, q_2), ..., (x_k, q_k), k ≥ 0. If path_i = q_k, then v_{ic} is a representative node. The pair (x_k, q_k) is thus unmarked in nestedpath (lines 9-10) which effectively removes the node-pair from the sequence. Now, every pair (v_i, q_i) that interlaces with the pair (v_i, low2_i) (i.e. low2_i ≤ q_i ≤ path_i) is unmarked in nestedpath (lines 12-13) as the node q_i cannot be a representative node owing to Theorem 4.3. Similarly, for each incoming non-tree link, (v_i, v_i),
of \( v_i \), every node-pair \((x_l, q_l)\) interlacing with \((v_{ij}, v_i)\) (i.e. \( \text{path}_i \preceq x_l \preceq \text{path}_{ij} \)) is unmarked in \text{nestedpath} (lines 15-18) because node \( q_l \) cannot be a representative node owing to Theorem 4.3. The \( q_l \)'s are also used to update \( \text{next}_i \) accordingly (line 18). Finally, on line 21, \( \text{path}_i \) and \( \text{next}_i \) are marked in \text{nestedpath}; the latter is then recorded into the field \text{nestedpath}_i.

**Lemma 6.3.** For every fair execution of Procedure \text{ComputeNestPath}, if there is a suffix \( \Pi \) in which \( \text{path}_i, \text{low1}_i, \text{low2}_i, \) and \( \text{lowchild}_i, 1 \leq i \leq n, \) contain the correct values in every configuration, then there is a suffix of \( \Pi \) in which, in every configuration, \text{nestedpath}_i contains the correct representation of \( S_{v_i}, 1 \leq i \leq n, \) and \( \text{next}_i \) contains the correct value.

**Proof:** After the execution reaches the first configuration of \( \Pi \), since \( \text{path}_i, \text{low1}_i, \text{low2}_i, \) and \( \text{lowchild}_i, 1 \leq i \leq n, \) contain the correct values, at every leaf node \( v_i \) of \( T_{dfs} \), the local variables \( \text{next}, \text{nestedpath}, \) and \( \text{lowchild} \) are correctly initialized to \( \text{low1}_i, \text{path}_i, \) and \( \text{lowchild}_i, \) respectively. Furthermore, as \( v_i \) is a leaf-node, \( \text{lowchild} \) must be \text{null}. As a result, \( \text{path}_i \) and \( \text{next}_i \) are correctly marked in \text{nestedpath} (which is \( \text{path}_i \)) and \( \text{next}_i \) is correctly set to \( \text{low2}_i \) on Line 21. Therefore, there is a suffix of \( \Pi \) in which \( \text{nestedpath}_i \) and \( \text{next}_i \) contain the correct values at every leaf-node \( v_i \) of \( T_{dfs} \).

Suppose there is a suffix \( \Pi' \) of \( \Pi \) in which, in every configuration, the fields \( \text{path}_j, \text{low1}_j, \text{low2}_j, \) \( \text{lowchild}_j, \) and \( \text{nestedpath}_j, 1 \leq j \leq n, \) contain the correct values at every node \( v_j \) on level \( h \) or higher (i.e. farther from \( v_1 \)). Let \( v_i \) be a node on level \( h - 1 \). If the \( \text{lowchild} \) of \( v_i \) is \text{null}, then \( \text{path}_i \) and \( \text{next}_i \) are marked in \text{nestedpath} (which is \( \text{path}_i \)) which correctly represents the sequence \( S_{v_i} : (v_{ij}, \text{low2}(v_i)) \) and \( \text{next}_i \) is correctly set to \( \text{low2}_i \) on Line 21. Otherwise, node \( v_i \) reads \( \text{nestedpath}_c \) from the \( \text{lowchild} v_{ic} \) and stores the value in the local variable \( \text{nestedpath} \). Since \( v_{ic} \) is on level \( h, \) \( \text{nestedpath} \) thus contains the correct representation of \( S_{v_{ic}} \). Node \( v_i \) then updates \( \text{nestedpath} \) by unmarking (removing) all those \((x_l, q_l)\) pairs that interlace with \((v_{ij}, \text{low2}_i)\) or with some incoming non-tree link, \((v_{ij}, v_i)\), of \( v_i \), and adding \((v_i, \text{next}_i)\) to \( \text{nestedpath} \) as the innermost pair. Node \( v_i \) also updates the local variable \( \text{next} \) to the smallest (w.r.t. \( \prec \)) \( \text{path}_{ql} \) of those \((x_l, q_l)\)'s that are unmarked, if \( \text{path}_{ql} \prec \text{low2}_l \) for some \( l \). Finally, the correct values of \( \text{nestedpath} \) and \( \text{next} \) are written into the fields \( \text{nestedpath}_i \) and \( \text{next}_i \), respectively.

Hence, there is a suffix of the execution in which, for every node \( v_i, 1 \leq i \leq n, \) \( \text{nestedpath}_i \) and \( \text{next}_i \) contain their correct values. \( \square \)
6.4 Computing 3-Edge-Connected Components

At every node $v_i, 1 \leq i \leq n$, a field $tcc_i$ is used to record the path value of $\text{reprenode}(v_i)$ and a field $rtcc_i$ is used to maintain an ordered list of the path values of the representative nodes that are ancestors of $v_i$ and are 3-edge-connected to some descendants of $v_i$. The $tcc$ values are generated at the representative nodes and are propagated downward in the $T_{dfs}$ through the $rtcc$ values.

Lemma 6.4. Let $v_i$ be a non-representative node and $v_j$ be the parent of $v_i$. Then $v_i$ and $v_j$ belong to different 3-edge-connected components if and only if $\text{next}_i$ is a proper prefix of $tcc_j$ (i.e. $\text{next}_i \prec tcc_j$).

Proof: Suppose $\text{next}_i \not\prec tcc_j$. Then the node-pairs $(v_j, tcc_j)$ and $(v_i, \text{next}_i)$ interlace. Since $v_j$ and $tcc_j$ are 3-edge-connected while $v_i$ and $\text{next}_i$ are 3-edge-connected (Lemma 5.3), by Corollary 4.3.2 and Lemma 2.1, nodes $v_i$ and $v_j$ are 3-edge-connected.

Suppose $\text{next}_i \prec tcc_j$. Then $\text{next}_i$ is a proper ancestor of $tcc_j$. Therefore, $tcc_j$ cannot be 3-edge-connected to $\text{next}_i$ as it is a representative node. But $\text{next}_i$ is 3-edge-connected to $v_i$ by Lemma 5.3. Hence, $v_j$ is not 3-edge-connected to $v_i$ owing to Lemma 2.1.

The above lemma shows that for every non-representative node $v_i$, if $\text{next}_i \not\prec tcc_j$, where $v_j$ is the parent node, then $tcc_i$ is $tcc_j$. Otherwise, owing to Corollary 4.3.2, $tcc_i$ must be the closest ancestor of $v_i$ in $rtcc_j$ excluding $tcc_j$.

Since $v_j$ is an ancestor of $v_i$ if and only if $\text{path}_j$ is a prefix of $\text{path}_i$, as with $\text{nestedpath}$, the ordered list of ancestors of $v_i$ in $rtcc_i$ can be represented in a compact form based on a path value which is either $\text{path}_i$ or the path value of the closest ancestor of $v_i$ in $rtcc_j$. Specifically, $rtcc_i$ consists of a path value with intervening $\$ symbols each corresponding to a distinct node in the list such that $v_j$ is the $h$-th element in the list if and only if the prefix of $rtcc_i$ terminating by the $h$-th $\$ symbol is $\text{path}_j$ (ignoring the intervening $\$ symbols). The following are the functions used in the Procedure below for inserting or removing $\$ symbols in $rtcc$ so as to trace the path values of representative nodes:

1. $\text{remove}\$(rtcc): Returns the path value after removing all the $\$ symbols from rtcc.

   For example, $\text{remove}\$(\alpha_1\$\alpha_2\alpha_3\$) = $\alpha_1\alpha_2\alpha_3$.

2. $\text{delnode}(rtcc)$: Returns the prefix of $rtcc$ that ends at (and including) the second rightmost $\$ symbol or returns the symbol $\perp$ if there is no second rightmost $\$. For example,
\[ \text{delnode}(\alpha_1 \alpha_2 \alpha_3 \alpha_4 \alpha_5 \alpha_6 \$) = \alpha_1 \alpha_2 \alpha_3 \alpha_4 \alpha_5 \$; \text{delnode}(\alpha_1 \alpha_2 \alpha_3 \alpha_4 \alpha_5 \alpha_6 \$) = \alpha_1 \alpha_2 \alpha_3 \$; \text{delnode}(\alpha_1 \alpha_2 \$) = \bot. \]

3. \text{track}(\text{path}, \text{rtcc}): Let \( \text{suf} \) be the suffix of \( \text{path} \) such that \( \text{remove}\$ (\text{rtcc}) \oplus \text{suf} = \text{path} \). Then \( \text{track}(\text{path}, \text{rtcc}) \) returns \( \text{rtcc} \oplus \text{suf} \oplus \$. For example, \( \text{track}(\alpha_1 \alpha_2 \alpha_3 \alpha_4 \alpha_5 \alpha_6 \alpha_7, \alpha_1 \alpha_2 \alpha_3 \alpha_4 \alpha_5 \$) = \alpha_1 \alpha_2 \alpha_3 \alpha_4 \alpha_5 \alpha_6 \alpha_7 \$. 

```
1 ComputeID(\( v_i, v_{ij}, 1 \leq j \leq \delta_i \)): /* Procedure for computing \( tcc_i, i \geq 2 \) */
2 begin
3    \( \text{next} := \text{read}(\text{next}_i); \text{path} := \text{read}(\text{path}_i); \text{nestedpath} := \text{read}(\text{nestedpath}_i); \)
4    for \( j := 1 \) to \( \delta_i \) do
5        if \( (\text{path} = \text{readpath}_i \oplus \alpha_{ij}(i)) \) then
6            \( \text{readrtcc}_j := \text{read}(\text{rtcc}_ij); \text{readtcc}_j := \text{remove}\$ (\text{readrtcc}_j); \) /* read \( \text{tcc}_{ij} \) */
7            if \( (\text{path} = \text{next}_i) \) then /* \( v_i \) is a representative node */
8                \( \text{tcc} := \text{next}_i; \) /* copy its \( \text{path} \) value into its \( \text{tcc} \) */
9                if \( (\exists (x_h, q_h) \) in \text{nestedpath} \) such that \( \text{path}_{q_h} = \text{readtcc}_j \) then
10                   \( \text{rtcc} := \text{track}(\text{path}, \text{readtcc}_j) \)
11                else /* \( v_i \) is not a representative node */
12                    \( \text{rtcc} := \text{track}(\text{path}, \text{delnode}(\text{readtcc}_j)); \) /* remove \( \text{tcc}_j \) */
13                end
14            end
15        else /* \( v_i, v_{ij} \) are of same component */
16            \( \text{tcc} := \text{readtcc}_j; \text{rtcc} := \text{readrtcc}_j; \) /* \( \text{tcc} \) of \( v_i, v_{ij} \) are same */
17            \( \text{rtcc} := \text{delnode}(\text{readtcc}_j); \text{tcc} := \text{remove}\$ (\text{rtcc}); \) /* extract \( \text{tcc} \) from \( \text{rtcc}_{ij} \) */
18        end
19    end
20    \( \text{write } tcc_i := \text{tcc}; \text{write } rtcc_i := \text{rtcc}; \) /* write \( \text{tcc} \) and \( \text{rtcc} \) into register */
21 end
```

Procedure ComputeID(\( v_i, v_{ij}, 1 \leq j \leq \delta_i \))

The root \( v_1 \) keeps the constant value \( \bot \) in \( tcc_1 \) and \( \bot \$\), representing the list consisting of \( v_1 \), in \( rtcc_1 \) (see the main algorithm, Algorithm 5 below). Every processor \( v_i, i \geq 2 \), calls Procedure ComputeID(\( v_i, v_{ij}, 1 \leq j \leq \delta_i \)) to compute \( tcc_i \) and \( rtcc_i \) as follows: The node \( v_i \) reads the \( rtcc_{ij} \) field of the parent node \( v_{ij} \) into its local variable \( \text{readrtcc}_j \) (Line 6).

If \( \text{next}_i = \text{path}_i \) (Line 7), then \( v_i \) is a representative node by Theorem 5.4. So, node \( v_i \) stores \( \text{path}_i \) in \( tcc_i \) (Line 8). Furthermore, if \( tcc_{ij} \) appears in a node-pair in \( \text{nestedpath}_i \) (Line 9), then it indicates that \( tcc_{ij} \) is 3-edge-connected to some descendant of \( v_i \). So \( v_i \) simply adds itself to the list \( rtcc_j \) and stores the resulting list in \( rtcc_i \) (Line 10). Otherwise, \( v_{ij} \) is removed from \( rtcc_j \) before \( v_i \) is added (Line 12).
If \( next_i \neq path_i \), then \( v_i \) is not a representative node. Moreover, if \( tcc_i \leq next_i \), then by Lemma 6.4, \( v_i \) and \( v_j \) belong to the same 3-edge-connected component. So, node \( v_i \) simply copies \( tcc_i \) and \( rtcc_i \) into \( tcc \) and \( rtcc \), respectively (Line 15). Otherwise, \( v_i \) and \( v_j \) belong to different 3-edge-connected components. Since \( v_j \) cannot be 3-edge-connected to any descendant of \( v_i \) owing to Corollary 4.3.2, it is thus removed from \( rtcc_j \) and the resulting list is then written into \( rtcc_i \) (Lines 17 and 20). Furthermore, as the second closest ancestor of \( v_i \) in \( rtcc_j \) is \( reprenode(v_i) \) which has become the closest ancestor of \( v_i \) in \( rtcc_i \), after all the intervening $ symbols are removed, is the desired \( tcc_i \) and is thus stored in \( tcc_i \) (Lines 17 and 20).

**Lemma 6.5.** For every fair execution of Procedure **ComputeID**, if there is a suffix \( \Pi \) in which \( path_i \), \( nestedpath_i \), \( next_i \), \( 1 \leq i \leq n \), contain the correct values in every configuration, then there is a suffix of \( \Pi \) in which, in every configuration, \( tcc_i = reprenode(v_i), 1 \leq i \leq n \).

**Proof:** Suppose the execution has reached a configuration in \( \Pi \). Then \( path_i, next_i, nestedpath_i, 1 \leq i \leq n \), contain the correct values. We shall apply induction to prove the following assertion:

For every integer \( l \geq 0 \), there is a suffix of \( \Pi \) in which, in every configuration, \( rtcc_i \) consists of an ordered list of representative nodes \( q_1, q_2, \cdots, q_k \) such that each \( q_j, 1 \leq j < k \), is 3-edge-connected to some descendants of \( v_i \) while \( q_k \) is 3-edge-connected to \( v_i \), and \( tcc_i = reprenode(v_i) \), for every node \( v_i \) on level \( h \leq l \) in the \( Tdfs \).

The root \( v_1 \) is the only node on level 0. Since \( v_1 \) always keeps the constant \( \bot \) and \( \bot \$ \) in \( tcc_1 \) and \( rtcc_1 \), respectively, the assertion clearly holds true for \( v_1 \).

Suppose the assertion holds for all nodes on level \( l \leq h \).

Let \( v_i \) be a node on level \( h + 1 \). Since \( v_i \) reads \( rtcc_i \) from its parent node, \( v_{ij} \), which is on level \( h \), by assumption, both \( rtcc_{ij} \) and \( tcc_{ij} \) satisfy the stated conditions. If \( path_i = next_i \), then \( path_i(= reprenode(v_i)) \) is correctly written into \( tcc_i \) on Line 8 since the values \( path_i \) and \( next_i \) are already correctly computed. Moreover, if there is a node-pair \( (x_m, q_m) \) in \( nestedpath_i \) where \( path_{q_m} = tcc_{ij} \), then \( tcc_{ij} \) is 3-edge-connected to some descendant of \( v_i \). Nodes \( v_i \) thus correctly executes Line 10 which adds \( tcc_i \) (which is \( path_i \)) to \( rtcc_{ij} \). Since \( rtcc_{ij} \) satisfies the condition given in the assertion, the resulting \( rtcc_i \) clearly satisfies the condition given in the assertion. On the other hand, if there is no such node-pair \( (x_m, q_m) \) in \( nestedpath_i \), then \( tcc_{ij} \) is not 3-edge-connected to any descendant of \( v_i \). Nodes \( v_i \) thus correctly executes Line 12 which removes \( tcc_{ij} \) from \( rtcc_{ij} \) before adding \( tcc_i \) to \( rtcc_{ij} \). Again, the resulting \( rtcc_i \) clearly satisfies the condition given in the assertion.
Algorithm 5: 3-EDGE-CONNECTIVITY

If \( path_i \neq \text{next}_i \), then by Theorem 5.4, \( v_i \) is not a representative node. Furthermore, if \( tcc_{ij} \leq \text{next}_i \), then by Lemma 6.4, \( v_i \) and \( v_{ij} \) belong to the same 3-edge-connected component. Node \( v_i \) thus correctly copies \( tcc_{ij} \) and \( rtcc_{ij} \) into \( tcc_i \) and \( rtcc_i \), respectively on Line 15. The assertion clearly holds true for node \( v_i \). On the other hand, if \( \text{next}_i < tcc_{ij} \), then \( v_i \) and \( v_{ij} \) belong to different 3-edge-connected components by Lemma 6.4. It then follows from Corollary 4.3.2 that no descendant of \( v_i \) can be 3-edge-connected to \( tcc_{ij} \). As a result, node \( v_i \) correctly removes \( tcc_{ij} \) from \( rtcc_{ij} \) and writes the resulting list into \( rtcc_i \) on Line 17. Again, by Corollary 4.3.2, \( tcc_i \) must be the closest ancestor of \( v_i \) in \( rtcc_i \). The \( path \) value of this ancestor is \( rtcc_i \) with all the $ symbols removed. Hence, \( tcc_i \) is correctly assigned the value \( \text{remove}\$)(rtcc_i) on Line 17. The assertion thus holds for all nodes on level \( h+1 \). □

6.5 The Self-Stabilizing Algorithm

The task of determining the 3-edge-connected components is defined by the set of legal executions in which, in every configuration, \( tcc_i = \text{reprenode}(v_i), 1 \leq i \leq n \).

The special processor \( v_1 \) (root) executes Lines 2 to 4 of Algorithm 3-EDGE-CONNECTIVITY. Since \( v_1 \) must be a representative node, it therefore repeatedly writes its \( path \) value (\( \perp \)) into the \( path_1 \) and \( tcc_1 \) fields and its lists of ancestors that are representative nodes (\( \perp $\)) into the \( rtcc_1 \) field.

Every non-root processor \( v_i, i \geq 2 \), executes Lines 7 to 12 of Algorithm 3-EDGE-CONNECTIVITY. The processor repeatedly calls Procedures \text{ComputePath}, \text{ComputeLow}, \text{ComputeNestedPath}, and \text{ComputeID} in that order for computing \( path_i, low_1 \) and \( low_2 \), \( nestedpath_i \), \( tcc_i \), respectively. In each procedure, processor \( v_i \) reads from its neighboring processors \( v_{ij} \), \( 1 \leq j \leq \delta_i \),
and writes into its own register. When the algorithm stabilizes, the path value of reprenode($v_i$) is
kept in $tcc_i$.

The $tcc$-value at every node uniquely identifies the 3-edge-connected component that contains
the node. Thus, upon stabilization, every node knows the identity of the 3-edge-connected com-
ponent containing it. The similar approach was followed to identify the 2-edge-connected components
in [3], where, upon stabilization, every node is labelled with an identifier that uniquely identifies
the 2-edge-connected component containing it.

**Theorem 6.6.** For every fair execution of Algorithm **3-EDGE-CONNECTIVITY**, there is
a suffix in which for every node $v_i$, $1 \leq i \leq n$, $tcc_i$ is the path value of reprenode($v_i$) in every
configuration.

**Proof:** The **3-EDGE-CONNECTIVITY** algorithm is developed by embedding new instructions
in the self-stabilizing depth-first search algorithm of Collin and Dolev [4]. These new instructions
do not affect the original function of the depth-first search algorithm. The depth-first search part
of the algorithm thus correctly constructs a depth-first search spanning tree $T_{dfs}$ of the network.
The theorem then follows from Lemma 6.1, Lemma 6.2, Lemma 6.3, and Lemma 6.5.

**Theorem 6.7.** Algorithm **3-EDGE-CONNECTIVITY** stabilizes in $O(dn\Delta)$ rounds and every
processor requires $O(n \log \Delta)$ bits, where $\Delta$ is an upper bound on the degree of a processor and $d$
is the diameter of the network.

**Proof:** It is easily verified that the new instructions added to the depth-first search algorithm of
Collin and Dolev [4] increase the time complexity for constructing a depth-first search tree only by
a constant factor. Therefore, the time required by Algorithm **3-EDGE-CONNECTIVITY** to
construct a depth-first search tree remains as $O(dn\Delta)$ rounds. The for loop for computing low1
and low2 requires $O(\Delta)$ rounds. The for loop for computing the nestedpath takes $O(\Delta)$ rounds,
the for loop for computing the tcc values takes $O(1)$ rounds. By applying an induction on the level
of the nodes in the spanning tree, it is easily verified that, once the $T_{dfs}$ is constructed, $O(H\Delta)$
rounds later, where $H(< n)$ is the height of $T_{dfs}$, every node $v_i$, $1 \leq i \leq n$, correctly determines the
path value of reprenode($v_i$).

In the depth-first search algorithm of Collin and Dolev [4], the space required by every processor
is $O(n \log \Delta)$ bits. This is the space required to store the path value of the processor. In Algorithm
**3-EDGE-CONNECTIVITY**, each of the fields path, low1, low2, tcc is also a path value and
its component identifier upwards to the representative node of the 3-edge-connected component for an undirected graph whose nodes are CONNECTIVITY.

Figure 3: A Depth-First Search Spanning Tree $T_{dfs}$. In this graph, cut-pairs are: \{(v_9, v_{10}), (v_{10}, v_1)\}; \{(v_9, v_{11}), (v_{11}, v_1)\}; \{(v_8, v_9), (v_1, v_2)\}; \{(v_7, v_8), (v_2, v_3)\}; \{(v_5, v_6), (v_4, v_5)\} and 3-edge-connected components are: \{v_{10}\}; \{v_{11}\}; \{v_9, v_1\}; \{v_8, v_2\}; \{v_7, v_6, v_4, v_3\}; \{v_5\}
each of the fields nestedpath and rtcc is at most twice the size of the largest path value. The space complexity per processor is thus $O(n \log \Delta)$ bits.

Figure 3 shows a depth-first search spanning tree $T_{dfs}$ constructed by Algorithm 3-EDGE-CONNECTIVITY for an undirected graph whose nodes are $v_1, v_2, \cdots, v_{11}$. For every node $v_i$, $2 \leq i \leq 11$, the figure also shows the values of low1i and low2i calculated by the algorithm. The sequence of ordered node-pairs represented by the field nestedpath at $v_{11}, v_{10}, v_9, v_8, v_7, v_6, v_5, v_4, v_3,$ and $v_2$ are: \{(v_{11}, v_{11})\}, \{(v_{10}, v_{10})\}, \{(v_9, v_1)\}, \{(v_9, v_1), (v_9, v_2)\}, \{(v_9, v_1), (v_8, v_2), (v_7, v_4)\}, \{(v_9, v_1), (v_8, v_2), (v_6, v_3)\}, \{(v_9, v_1), (v_8, v_2), (v_6, v_3), (v_5, v_5)\}, \{(v_9, v_1), (v_8, v_2), (v_4, v_3)\}, \{(v_9, v_1), (v_8, v_2), (v_3, v_3)\}, and \{(v_9, v_1), (v_2, v_2)\}, respectively. When the algorithm stabilizes, tcc1 = tcc9 = path1; tcc10 = path10; tcc11 = path11; tcc5 = path5; tcc2 = tcc8 = path2; tcc3 = tcc4 = tcc6 = tcc7 = path3. Therefore, the 3-edge-connected components are: \{v_1, v_9\}, \{v_{10}\}, \{v_{11}\}, \{v_5\}, \{v_2, v_8\}, and \{v_3, v_4, v_6, v_7\}.

The algorithm can be easily modified so that each node knows the set of nodes that are 3-edge-connected to it. This can be done as follows: every node passes its identifier along with its component identifier upwards to the representative node of the 3-edge-connected component.
containing it. The representative node then broadcasts the identifiers of all the nodes that are 3-edge-connected to it downwards to all other nodes in the same 3-edge-connected component. This process can be completed in \( O(dn) \) rounds. The time complexity of the algorithm thus remains unchanged. The space complexity, however, becomes \( O(n^2 \log \Delta) \) per processor. This is because a node can be 3-edge-connected to \( O(n) \) other nodes and storing the identifiers of these \( O(n) \) nodes will require \( O(n^2 \log \Delta) \) space at the node.

7 Conclusion

We have presented a self-stabilizing algorithm for the 3-edge-connectivity problem. The algorithm constructs a depth-first search tree in \( O(dn\Delta) \) rounds and determines the 3-edge-connected components based on the depth-first search tree in \( O(H\Delta) \) additional rounds, where \( H < n \). Clearly, our algorithm will work correctly if the first depth-first search spanning tree is replaced by another type of depth-first search spanning tree. Therefore, the time complexity of our algorithm is actually \( \max\{T_{dfs}(n), O(H\Delta)\} \), where \( T_{dfs}(n) \) is the time complexity of the self-stabilizing algorithm that is used to construct the depth-first spanning tree. Since \( T_{dfs}(n) = O(dn\Delta) \) for the algorithm of Collin et al. [4], we thus have the time bound \( O(dn\Delta) \). Should there be an improvement made on \( T_{dfs}(n) \), the time complexity of our algorithm will automatically be improved.

Although our algorithm is designed for read/write atomicity, it had been pointed out that any algorithm designed to work in read/write atomicity also works in any system that has a central or distributed scheduler (daemon) [8]. Therefore, our algorithm also works under a distributed scheduler. Although we assume that the given network is bridgeless, our algorithm, with slight modifications, will also work for network with bridges. This is based on the observation that the intersection of the depth-first search tree with each bridge-connected component is a depth-first search tree of that bridge-connected component. Therefore, the 3-edge-connected components belonging to the same bridge-connected component can be generated after the construction of the depth-first search tree within that bridge-connected component stabilizes.

References


A. Proof of Lemma 5.1

Proof: By Lemma 4.2, there exists a non-tree link, \((z,\text{low}1(u))\), on the to-low path of \(u\). By assumption, the link \((z,\text{low}1(u))\) does not form a cut-pair with the parent link of \(u\). Therefore, by Theorem 4.3, there exists a non-tree link \((s,t)\) such that \(t \prec u \preceq s\). If \(s\) is not also a descendant of the lowchild of \(u\) or \(u\) has no lowchild, then we immediately obtain \(\text{low}2(u) \prec u\).
Suppose $u$ has a lowchild and $s$ is a descendant of the lowchild of $u$ for any of the aforementioned $(s, t)$ non-tree links. Then $\text{low}2(u) = u$. Of all these $(s, t)$ links, we choose one for which the closest ancestor of $s$ on the to-low path of $u$ is closest to $u$ and let $w_1$ be that ancestor of $s$. This implies that there does not exist a node $w(\neq w_1)$ on the $u - w_1$ tree-path such that $\text{low}2(w) < u$. It follows that there is no non-tree link $(s, t)$ such that $t < u$ while $u \preceq s$ and $w_1 \not\preceq s$. Moreover, $\text{low}2(w_1) \prec u \prec w_1$.

Since by assumption, the parent link of $u$ and the parent link of $w_1$ do not form a cut-pair, there must exist a non-tree link $(s, t)$ such that $u \preceq t < w_1$ while $w_1 \preceq s$. Let $(a, b)$ be one of these $(s, t)$ links such that $b$ is closest to $u$ and let the closest ancestor of $a$ on the to-low path of $u$ be $w_2$. If $b = u$, we have the desired sequence.

Suppose $b \neq u$. Since the parent link of $u$ and the parent link of $b$ do not form a cut-pair, by Theorem 4.3, there must exist a non-tree link $(s, t)$ such that $u \preceq t < b$ while $b \preceq s$. Let $w$ be the closest ancestor of $s$ on the to-low path of $u$. Then $\text{low}2(w) < b$. Moreover, from the way we determine $w_2$, node $w$ must lie on the $b - w_1$ tree-path, excluding $w_1$. Now, of all the aforementioned $w$ nodes, let $w_3$ be one such that $\text{low}2(w_3)$ is closest to $u$. Then $\text{low}2(w_3) < b$.

If $\text{low}2(w_3) = u$, we have the desired sequence. Otherwise, as the parent link of $u$ and the parent link of $\text{low}2(w_3)$ do not form a cut-pair, by Theorem 4.3, there must exist a non-tree link $(s, t)$ such that $u \preceq t < \text{low}2(w_3)$ while $\text{low}2(w_3) \preceq s$. Let $w$ be the closest ancestor of $s$ on the to-low path of $u$. Then $\text{low}2(w) < \text{low}2(w_3)$. Moreover, by the way we determine $w_i, 2 \leq i \leq 3$, node $w$ must lie on the $\text{low}2(w_3) - b$ tree-path, excluding $b$. Now, of all the aforementioned $w$ nodes, let $w_4$ be one such that $\text{low}2(w_4)$ is closest to $u$. Then $\text{low}2(w_4) < \text{low}2(w_3)$.

If $\text{low}2(w_4) = u$, we have the desired sequence. Otherwise, as the parent link of $u$ and the parent link of $\text{low}2(w_4)$ do not form a cut-pair, by Theorem 4.3, there must exist a non-tree link $(s, t)$ such that $u \preceq t < \text{low}2(w_4)$ while $\text{low}2(w_4) \preceq s$. Let $w$ be the closest ancestor of $s$ on the to-low path of $u$. Then $\text{low}2(w) < \text{low}2(w_4)$. Moreover, from the way we determine $w_i, 2 \leq i \leq 4$, node $w$ must lie on the $\text{low}2(w_4) - \text{low}2(w_3)$ tree-path, excluding $\text{low}2(w_3)$. Now, of all the aforementioned $w$ nodes, let $w_5$ be one such that $\text{low}2(w_5)$ is closest to $u$. Then $\text{low}2(w_5) < \text{low}2(w_4)$.

If $\text{low}2(w_5) = u$, we have the desired sequence. Otherwise, by repeating the above argument, we will obtain the desired sequence.

\[\square\]

B. Proof of Theorem 5.2
Proof: Suppose the parent link of $u$ does not form a cut-pair with any link in $T(u)$ or having an end-node in $T(u)$. By Lemma 5.1, either $low^2(u) \prec u$ or there exists a sequence of nodes $w_i, 1 \leq i \leq k$, on the to-low path of $u$ satisfying Conditions (i)-(vi). In the former case, $low^2(u)$ is not a descendant of $u$ that is 3-edge-connected to $u$ owing to Lemma 4.1. In the latter case, as $low^2(w_1) \prec b \preceq w_1 \preceq w_2$ (Figure 2) and $w_1$ and $low^2(w_1)$ are 3-edge-connected owing to Lemma 4.1, by Corollary 4.3.2, nodes $b$ and $w_1$ are 3-edge-connected. But then, by Lemma 2.1, $low^2(w_1)$ and $b$ are 3-edge-connected. Similarly, as $low^2(w_3) \prec b \preceq w_3 \preceq w_2$, nodes $b$ and $w_3$ are 3-edge-connected which implies that $b$ and $low^2(w_3)$ are 3-edge-connected. For $i, 4 \leq i \leq k$, since $low^2(w_i) \prec low^2(w_{i-1}) \preceq w_i \prec w_{i-1}$ and $w_i$ is 3-edge-connected to $low^2(w_i)$, nodes $low^2(w_{i-1})$ and $w_i$ are 3-edge-connected which implies that $low^2(w_i)$ and $low^2(w_{i-1})$ are 3-edge-connected. It then follows from Lemma 2.1 that $low^2(w_1)$ is 3-edge-connected to $low^2(w_k) = u$. As $low^2(w_1) \prec u$, node $u$ is thus 3-edge-connected to a non-descendant node.

Suppose the parent link of $u$ forms a cut-pair with a link in $T(u)$ or having an end-node in $T(u)$. By Theorem 4.3, it is easily verified that for any node $w$ outside $T(u)$, there are at most two edge-disjoint paths connecting $u$ and $w$: one passing through the parent link of $u$; the other passing through the link that forms the cut-pair with the parent link of $u$. Hence, all the nodes that are 3-edge-connected to $u$ are in $T(u)$. \hfill \Box

C. Proof of Lemma 5.3

Proof: If $u$ is a leaf node, then $next(u) = low^2(u)$. By Lemma 4.1, node $u$ is 3-edge-connected to $next(u)$. Let $u$ be a non-leaf node. Suppose $next(w)$ is 3-edge-connected to $w$, for every proper descendant $w$ of $u$. If $u$ has no incoming non-tree link interlacing with an $(x_j, q_j)$ in the nested sequence of its lowchild such that $q_j \prec low^2(u)$, then $next(u) = low^2(u)$ which implies that $next(u)$ is 3-edge-connected to $u$. Otherwise, let $f$ be the smallest index such that $(x_f, q_f)$ interlaces with some $(s, u) \in In(u)$. Then $next(u) = q_f$. Since $x_f$ is a proper descendant of $u$, $next(x_f)$ is 3-edge-connected to $x_f$ by assumption. By Corollary 4.3.2, $u$ is 3-edge-connected to $x_f$. But $next(x_f) = q_f$; therefore $x_f$ is 3-edge-connected to $q_f$. By Lemma 2.1, $u$ is 3-edge-connected to $q_f$ which is $next(u)$. The lemma thus follows. \hfill \Box

D. Proof of Theorem 5.4
**Proof:** Suppose \( u \) is not a representative node. By Theorem 5.2, the parent link of \( u \) does not form a cut-pair with any link in \( T(u) \) or having an end-node in \( T(u) \). By Lemma 5.1, \( low(2)(u) \prec u \) or there exists a sequences of nodes \( w_i, 1 \leq i \leq k \), on the to-low path of \( u \) satisfying Conditions (i)-(vi). In the former case, as \( next(u) \preceq low(2)(u) \) by definition, we thus have \( next(u) \prec u \).

In the latter case, \( low(2)(w_1) \prec u \) implies that \( next(w_1) \prec u \). Since the non-tree link \((a,b)\) interlaces with the node-pair \((w_1,low(2)(w_1))\), it must interlace with either \((w_1,next(w_1))\) or a node-pair \((x_j,q_j)\) such that \( x_j \) lies on the \( b-w_1 \) tree-path while \( q_j \preceq next(w_1) \). It follows that \( next(b) \preceq next(w_1) \) which implies that \( next(b) \prec u \). Similarly, as \((w_3,low(2)(w_3))\) interlaces with \((b,next(b))\), \( next(low(2)(w_3)) \preceq next(b) \) which implies that \( next(low(2)(w_3)) \prec u \). For \( i,4 \leq i \leq k \), as \((w_i,low(2)(w_i))\) interlaces with \((low(2)(w_{i-1}),next(low(2)(w_{i-1})))\), \( next(low(2)(w_i)) \preceq next(low(2)(w_{i-1})) \).

But \( next(low(2)(w_{i-1})) \prec u \). Therefore, \( next(low(2)(w_i)) \prec u \). Since \( low(2)(w_k) = u \), we thus have \( next(u) \prec u \).

Suppose \( u \) is a representative node. By Theorem 5.2, the parent link of \( u \) forms a cut-pair with some link in \( T(u) \) or having an end-node in \( T(u) \). We shall consider the first case as the second case is similar but simpler. Let the link be \((v,w)\). By Theorem 4.3, there is no \((s,v) \in In(v)\) such that \( w \preceq s \), and there is no non-tree link \((s,t)\) such that \( t \prec u \) while \( v \preceq s \) and \( w \not\preceq s \). It follows that there is no incoming non-tree link of \( v \) interlacing with some node-pair in \( S_w \) and \( u \preceq low(2)(v) \). As a result, \( u \preceq next(v) \). Let \( x \) be any proper ancestor of \( v \) on the \( u-v \) tree-path such that \( u \preceq next(y) \) for every proper descendant \( y \) of \( x \) lying on the \( u-v \) tree-path. Then as with \( v \), \( u \preceq low(2)(x) \) and there is no \((s',x) \in In(x)\), such that \( w \preceq s' \). It follows that every incoming non-tree link of \( x \) can only interlace with node pairs \((x_j,q_j)\) in the sequence of the lowchild of \( x \) such that \( x_j \) lies on the \( x-v \) tree-path. Let \( f \) be the smallest index such that \( (x_f,q_f) \) interlaces with some incoming non-tree link of \( x \). Then \( u \preceq next(x_f) \) as \( x_f \) is one of the \( y \) vertices. But \( next(x_f) = q_f \). Therefore, \( u \preceq q_f \). It follows that \( u \preceq \min_{z} \{low(2)(x),q_f\} = next(x) \). When \( x = u \), we have \( u \preceq next(u) \).

Since \( next(u) \preceq u \) by definition, we thus have \( next(u) = u \). \( \square \)

**E. Proof of Theorem 6.4**

**Proof:** Suppose \( next_i \not\prec tcc_j \). Then the node-pairs \((v_j,tcc_j)\) and \((v_i,next_i)\) interlace. Since \( v_j \) and \( tcc_j \) are 3-edge-connected while \( v_i \) and \( next_i \) are 3-edge-connected (Lemma 5.3), by Corollary 4.3.2 and Lemma 2.1, nodes \( v_i \) and \( v_j \) are 3-edge-connected.

Suppose \( next_i \prec tcc_j \). Then \( next_i \) is a proper ancestor of \( tcc_j \). Therefore, \( tcc_j \) cannot be
3-edge-connected to $next_i$ as it is a representative node. But $next_i$ is 3-edge-connected to $v_i$ by Lemma 5.3. Hence, $v_j$ is not 3-edge-connected to $v_i$ owing to Lemma 2.1.  