Report Number: wucse-2009-8

2009

Self-Stabilizing Computation of 3-Edge-Connected Components

Abusayeed Saifullah and Yung Tsin

2009-8

Self-Stabilizing Computation of 3-Edge-Connected Components

Authors: Abusayeed Saifullah and Yung Tsin

Corresponding Author: saifullaha@cse.wustl.edu

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

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

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 stabilizes in $O(dn\Delta)$ rounds and each of the other two stabilizes in $O(n)$ rounds in the worst case. The first algorithm of the composition constructs a special spanning tree of the system, called a *first depth-first search tree*, based on the self-stabilizing depth-first search algorithm of Collin et al. [4]. In the second algorithm, every 3-edge-connected component is computed at a specific node that belongs to the component. The third algorithm is dedicated to propagate the results computed in the second phase to every other node. Thus, multiple phases comprise the 3-edge-connectivity algorithm where the computation of every phase, except the first phase, depends on the results of the preceding phase. The space complexity of the algorithm is $O(n^2 \log \Delta)$ bits per processor. The drawbacks of a composite self-stabilizing algorithm have been explained by Tsin [20]. Specifically, the time complexity of a composite algorithm, in the worst case, is the *product* of the time complexities of the algorithms that make up the composite algorithm if the time complexity is measured in terms of *step*. If the time complexity is measured in terms of *round*, although the time complexity of the composite algorithm is the *sum* of the time complexities of the algorithms that make up the composite algorithm, however, each *round* of the composite algorithm will be a *non-constant factor* larger than the *round* of the non-composite algorithm that solves the same problem. This non-constant factor is the product of the time complexities of those algorithms that must run concurrently with the algorithm.

In this paper, we present a self-stabilizing algorithm, with time and space complexity substantially improved, for 3-edge-connectivity of an asynchronous distributed computer network. The algorithm is non-composite in the sense that it does not require other self-stabilizing algorithms running concurrently with it. The time complexity of the algorithm is $O(dn\Delta)$ rounds and every processor requires only $O(n \log \Delta)$ bits. Note that these time and space complexity are dominated by the part of the algorithm that constructs a depth-first search spanning tree based on the algorithm of Collin et al. [4]. The remaining part of the algorithm that determines the 3-edge-connected components takes only $O(n\Delta)$ rounds. In other words, the time complexity of the algorithm is $\max\{T_{dfs}(n), O(n\triangle)\}$, where $T_{dfs}(n)$ is the time complexity for constructing a depth-first search tree for a network of n nodes. Since $T_{dfs}(n) = O(dn\Delta)$ for the algorithm of Collin et al. [4], the stated $O(dn\Delta)$ time bound thus follows. Hence, any improvement made in the time or space complexity for constructing a depth-first search tree will automatically imply an improvement in our algorithm. When the algorithm stabilizes, each processor is assigned a component identifier to uniquely identify the 3-edge-connected component to which the processor belongs.

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 ***link***s. Two nodes are ***neighboring*** if they are connected by an edge and the two nodes are the ***end-node***s 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 $Des(u)$. *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. ***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.

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* $\{v_1, v_2, \cdots, v_n\}$, 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. 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 $2, \cdots, n$ 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 α_i. For any edge $e = (v_i, v_j)$, $\alpha_i(j)$ ($\alpha_j(i)$, respectively) denotes the *edge index* of e according to α_i (α_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 χ_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 $\Re = (c_0, c_1, \cdots, c_i, c_{i+1}, \cdots)$ 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 $(c_0, c_1, \cdots, c_{k-1})$ 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 $\Re$ is the shortest prefix of $\Re$ in which every processor executes at least one step. Let $\Re = \Re_1\Re_2$ such that $\Re_1$ is the prefix consisting of the first k rounds of $\Re$. Then the $(k+1)$-th round of $\Re$ is the first round of $\Re_2$.

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

Definition 2. $$low2(v) = \begin{cases} \boldsymbol{min}(\{low1(x) | x \in C(v) - \{w\}\} \cup \{dfs(s) | (v,s) \in Out(v)\} \cup \{dfs(v)\}), \\ \qquad\qquad \text{if } \exists w \in C(v) \text{ such that } low1(v) = low1(w); \\ \boldsymbol{min}(\{low1(x) | x \in C(v)\} \cup \{dfs(s) | (v,s) \in Out(v) \wedge s \neq low1(v)\} \cup \\ \{dfs(v)\}), \qquad \text{otherwise}; \end{cases}$$

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 $w \in C(v)$ and $low1(w) = low1(v)$, or $(v, w) \in Out(v)$ and $dfs(w) = low1(v)$. In the former case, node w is called a* ***lowchild*** *of v.*

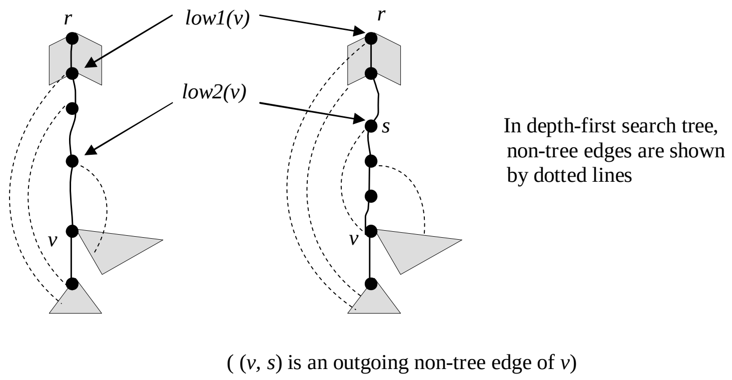

Figure 1: An illustration of $low1(v)$ and $low2(v)$ for $v \in V$ (In the figure depth-first search starts from the node r)

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 the $v - low1(v)$ path in which every link is a tree link except the last one.*

Proof: By induction on the length of the *to-low path.* □

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]. □

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* $low2(u) \prec u$ *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) $low2(w_1) \prec u \prec w_1$*;*

(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) $b \preceq w_3 \prec w_1$ *and* $low2(w_3) \prec b$*;*

(iv) $low2(w_3) \preceq w_4 \prec b$ *and* $low2(w_4) \prec low2(w_3)$*;*

(v) $low2(w_{i-1}) \preceq w_i \prec low2(w_{i-2}), 5 \leq i \leq k,$ *and* $low2(w_i) \prec low2(w_{i-1}), 5 \leq i \leq k$*;*

(vi) $low2(w_k) = u$*.*

Proof: By Lemma 4.2, there exists a non-tree link, $(z, low1(u))$, on the *to-low* path of u. By assumption, the link $(z, low1(u))$ 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 $low2(u) \prec u$.

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 $low2(u) = u$. 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 $low2(w) \prec u$. It follows that there is no non-tree link (s,t) such that $t \prec u$ while $u \preceq s$ and $w_1 \not\preceq s$. Moreover, $low2(w_1) \prec u \prec w_1$.

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 $u \preceq t \prec w_1$ 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 $u \preceq t \prec b$ while $b \preceq s$. Let w be the closest ancestor of s on the *to-low* path of u. Then $low2(w) \prec b$. 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 $low2(w_3)$ is closest to u. Then $low2(w_3) \prec b$.

If $low2(w_3) = u$, we have the desired sequence. Otherwise, as the parent link of u and the parent link of $low2(w_3)$ do not form a cut-pair, by Theorem 4.3, there must exist a non-tree link (s,t) such that $u \preceq t \prec low2(w_3)$ while $low2(w_3) \preceq s$. Let w be the closest ancestor of s on the *to-low* path of u. Then $low2(w) \prec low2(w_3)$. Moreover, by the way we determine $w_i, 2 \leq i \leq 3$, node w must lie on the $low2(w_3) - b$ tree-path, excluding b. Now, of all the aforementioned w nodes, let w_4 be one such that $low2(w_4)$ is closest to u. Then $low2(w_4) \prec low2(w_3)$.

If $low2(w_4) = u$, we have the desired sequence. Otherwise, as the parent link of u and the parent link of $low2(w_4)$ do not form a cut-pair, by Theorem 4.3, there must exist a non-tree link (s,t) such that $u \preceq t \prec low2(w_4)$ while $low2(w_4) \preceq s$. Let w be the closest ancestor of s on the *to-low* path of u. Then $low2(w) \prec low2(w_4)$. Moreover, from the way we determine $w_i, 2 \leq i \leq 4$, node w must lie on the $low2(w_4) - low2(w_3)$ tree-path, excluding $low2(w_3)$. Now, of all the aforementioned w nodes, let w_5 be one such that $low2(w_5)$ is closest to u. Then $low2(w_5) \prec low2(w_4)$.

If $low2(w_5) = u$, we have the desired sequence. Otherwise, by repeating the above argument, we will obtain the desired sequence. □

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: 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 $low2(u) \prec u$ 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, $low2(u)$

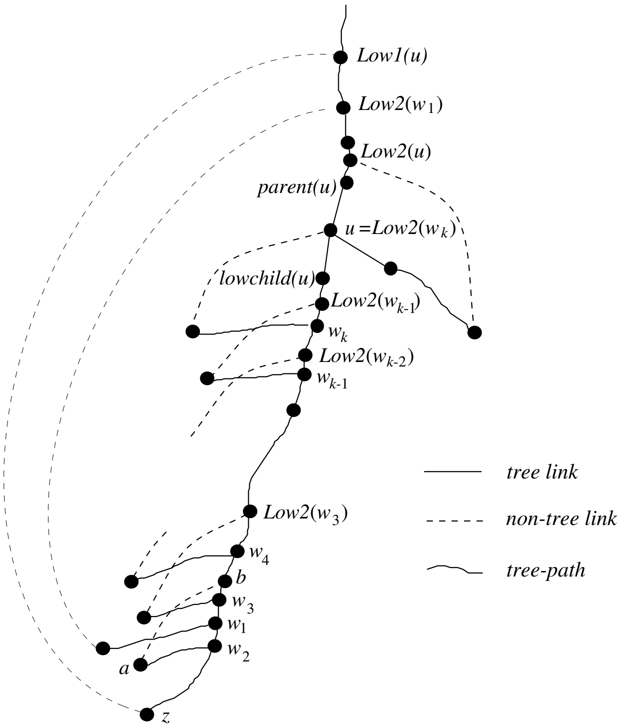

Figure 2: An Illustration of Lemma 5.1

is not a descendant of u although it is 3-edge-connected to u owing to Lemma 4.1. In the latter case, as $low2(w_1) \prec b \prec w_1 \preceq w_2$ (Figure 2) and w_1 and $low2(w_1)$ 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, $low2(w_1)$ and b are 3-edge-connected. Similarly, as $low2(w_3) \prec b \preceq w_3 \prec w_2$, nodes b and w_3 are 3-edge-connected which implies that b and $low2(w_3)$ are 3-edge-connected. For $i, 4 \leq i \leq k$, since $low2(w_i) \prec low2(w_{i-1}) \preceq w_i \prec w_{i-1}$ and w_i is 3-edge-connected to $low2(w_i)$, nodes $low2(w_{i-1})$ and w_i are 3-edge-connected which implies that $low2(w_i)$ and $low2(w_{i-1})$ are 3-edge-connected. It then follows from Lemma 2.1 that $low2(w_1)$ is 3-edge-connected to $low2(w_k) = u$. As $low2(w_1) \prec u$, 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 $T(u)$. □

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 **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 $reprenode(u)$ at every node u.

Definition 6. *A sequence of ordered pairs of nodes,* $(x_1, q_1), (x_2, q_2), \cdots, (x_k, q_k)$*, 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, $S_v : (x_1, q_1), (x_2, q_2), \cdots, (x_k, q_k)$, 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, $S_w : (x_1, q_1), (x_2, q_2), ..., (x_k, q_k)$, $k \geq 0$, 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 $next(v)$ which is defined as:

Definition 8.

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

$\boldsymbol{next}(v) = \boldsymbol{min}_{\prec}(\{low2(v)\} \cup \{q_j | \exists (s, v) \in In(v) \text{ such that } (x_j, q_j) \text{ interlaces with } (s, v)\})$.

Specifically, $next(v)$ is either the node $low2(v)$ 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 $next(v)$. 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 $next(v) = v$, then node v is the *representative node* of a 3-edge-connected component (see Theorem 5.4 below). Otherwise, if $next(v) = q_f$, then the node-pair (x_f, q_f) is replaced by

$(v, next(v))$, and if $next(v) \prec q_f$, then $(v, next(v))$ is simply added to S_w as the innermost node-pair. This is because the parent link of $next(v)$ 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, $next(u)$ is 3-edge-connected to u.*

Proof: If u is a leaf node, then $next(u) = low2(u)$. 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 $q_j \prec low2(u)$, then $next(u) = low2(u)$ 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 $(s, v) \in In(v)$. 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. □

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

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, $low2(u) \prec u$ 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 $next(u) \preceq low2(u)$ by definition, we thus have $next(u) \prec u$.

In the latter case, $low2(w_1) \prec u$ implies that $next(w_1) \prec u$. Since the non-tree link (a, b) interlaces with the node-pair $(w_1, low2(w_1))$, 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 $(w_3, low2(w_3))$ interlaces with $(b, next(b))$, $next(low2(w_3)) \preceq next(b)$ which implies that $next(low2(w_3)) \prec u$. For $i, 4 \leq i \leq k$, as $(w_i, low2(w_i))$ interlaces with $(low2(w_{i-1}), next(low2(w_{i-1})))$, $next(low2(w_i)) \preceq next(low2(w_{i-1}))$. But $next(low2(w_{i-1})) \prec u$. Therefore, $next(low2(w_i)) \prec u$. Since $low2(w_k) = u$, 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 $T(u)$. Let the link be (v, w). By Theorem 4.3, there is no $(s, v) \in In(v)$ such that $w \preceq s$ if w is the *lowchild* of v, and there is no non-tree link (s, t) such that $t \prec u$ while $u \preceq s$ 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 $u \preceq low2(v)$. As a result, $u \preceq next(v)$. Let x be a proper ancestor of v on the $u - v$ tree-path. Suppose $u \preceq next(y)$ for every proper descendant y of x lying on the $u - v$ tree-path. As with v, $u \preceq low2(x)$ 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 $u \preceq next(x_f)$ by assumption. But $next(x_f) = q_f$. Therefore, $u \preceq q_f$. It follows that $u \preceq \min_{\prec}\{low2(x), q_f\} = next(x)$. When $x = u$, we have $u \preceq next(u)$.

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

6 Adoption of Self-Stabilization

Since our algorithm is based on depth-first search, we shall use 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 $path_i$, in its register. At any point of time during the execution of the algorithm, $path_i$ 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, $path_i \prec path_j$ if and only if $path_j = path_i \oplus s$, 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 $path_1$ 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 *path* values from the registers of their neighboring processors and store the calculated result in the *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 $path_j \prec path_i$, and v_j is the *parent* node of v_i if v_j is the unique neighbor of v_i such that $path_i = path_j \oplus \alpha_j(i)$, $path_i$ can play the role of $dfs(v_i)$. The definitions of $low1$ and $low2$ can thus be rewritten as follows (where function $\boldsymbol{min}_{\prec}$ returns the lexicographically minimum *path*): $\forall v_i$, $1 \leq i \leq n$,

Definition 9. $low1_i = \mathbf{min}_{\prec}(\{path_i\} \cup \{low1_j | v_j \in C(v_i)\} \cup \{path_j | (v_i, v_j) \in Out(v_i)\})$;

Definition 10. $$low2_i = \begin{cases} \mathbf{min}_{\prec}(\{low1_j | v_j \in C(v_i) - \{v_k\}\} \cup \{path_j | (v_i, v_j) \in Out(v_i)\} \cup \{path_i\}), \\ \qquad if\ \exists v_k \in C(v_i)\ such\ that\ low1_i = low1_k; \\ \mathbf{min}_{\prec}(\{low1_j) | v_j \in C(v_i)\} \cup \{path_j | (v_i, v_j) \in Out(v_i) \wedge path_j \neq low1_i\} \cup \\ \{path_i\}), \quad otherwise. \end{cases}$$

The ***degree of a node*** v_i, denoted by δ_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 $path_i, path_j, \alpha_i(j)$, and $\alpha_j(i)$ as follows:

- The link (v_j, v_i) is the ***parent link*** if and only if $path_i = path_j \oplus \alpha_j(i)$;
- The link (v_i, v_j) is a ***child link*** if and only if $path_j = path_i \oplus \alpha_i(j)$;
- The link (v_i, v_j) is an ***outgoing non-tree edge*** if and only if $(\exists s)((path_i = path_j \oplus s) \wedge (s \neq \alpha_j(i)))$;
- The link (v_j, v_i) is an ***incoming non-tree edge*** if and only if $(\exists s)((path_j = path_i \oplus s) \wedge (s \neq \alpha_i(j)))$.

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 $low1, low2$, and $next$ based on the depth-first search tree, henceforth denoted by T_{dfs}, constructed by their algorithm.

Along with the $path_i$ field, every processor v_i, $i \geq 2$, maintains some additional fields: $low1_i$ (Definition 9), $low2_i$ (Definition 10), $nestedpath_i$, $next_i$, $rtcc_i$, tcc_i in its register. The special processor v_1 (root) maintains only the fields $rtcc_1$ and tcc_1 in addition to $path_1$. When the algorithm stabilizes, at every processor $v_i, 1 \leq i \leq n$, the field tcc_i contains the $path$ value of $reprenode(v_i)$.

The field $nestedpath_i$ is used to represent the node-pair sequence, $S_{v_i} : (x_1, q_1), (x_2, q_2), \cdots$, (x_k, q_k), defined in the previous section. However, instead of representing the sequence with a sequence of $2k$ $path$ values, we shall represent the sequence in a compact form, using only one $path$ 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, $path$, such that $x_1 \preceq path$. The content of $nestedpath_i$ 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_i}, 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,

$path_{x_1} = \alpha_1\alpha_2\alpha_3\alpha_4\alpha_5\alpha_6\alpha_7\alpha_8\alpha_9\alpha_{10}$ (the subsequence of indices up to the last \$ symbol);

$path_{q_1} = \alpha_1\alpha_2$ (the subsequence of indices up to the 1st \$ symbol);

$path_{x_2} = \alpha_1\alpha_2\alpha_3\alpha_4\alpha_5\alpha_6\alpha_7\alpha_8$ (the subsequence of indices up to the 2nd last \$ symbol);

$path_{q_2} = \alpha_1\alpha_2\alpha_3$ (the subsequence of indices up to the 2nd \$ symbol);

$path_{x_3} = \alpha_1\alpha_2\alpha_3\alpha_4\alpha_5\alpha_6\alpha_7$ (the subsequence of indices up to the 3rd last \$ symbol);

$path_{q_3} = \alpha_1\alpha_2\alpha_3\alpha_4\alpha_5$ (the subsequence of indices up to the 3rd \$ symbol).

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}, $1 \leq j \leq \delta_i$ (the degree of v_i), be the neighboring processors of processor v_i, $1 \leq i \leq n$, 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 $\mathbf{trunc}_N$ 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 $path_i$ are available in [4].

ComputePath(v_i, v_{ij}, $1 \le j \le \delta_i$): /* Procedure for computing $path_i, i \ge 2$ */
begin
  for $j := 1$ *to* δ_i **do** $readpath_j :=$ **read**($path_{ij}$); /* read the *path* value of neighbor v_{ij} into local variable $readpath_j$ */
  write $path_i :=$**min**$_\prec\{$**trunc**$_N(readpath_j \oplus \alpha_{ij}(i)) | 1 \le j \le \delta_i\}$; /* compute $path_i$ */
end

Procedure ComputePath(v_i, v_{ij}, $1 \le j \le \delta_i$)

Lemma 6.1. *For every fair execution of Procedure* ***ComputePath****, given that $path_1 = \perp$ in every configuration, where v_1 is the root of T_{dfs}, there is a suffix Π in which, in every configuration, $path_i, 1 \le i \le n$, is the smallest path connecting v_1 and node v_i.*

Proof: Immediate from Theorem 3.3 in [4]. □

6.2 Computing $low1_i$, $low2_i$

Every processor v_i, $i \ge 2$, calls the procedure **ComputeLow**(v_i, v_{ij}, $1 \le j \le \delta_i$) for computing $low1_i$ and $low2_i$. Each leaf node v_i computes $low1_i$ and $low2_i$ based on the *path* values it reads from its outgoing non-tree links (lines 11-14). Each non-leaf node v_i computes $low1_i$ and $low2_i$ based on the $low1$ values it reads from its children (lines 6-10) and the *path* values it reads from its outgoing non-tree links (lines 11-14). The procedure also records a *lowchild* (Definition 4) for v_i. If $low1_i$ is defined from some outgoing non-tree link of v_i, then $lowchild_i$ is recorded as *null* meaning that v_i has no *lowchild* (lines 12-13).

Lemma 6.2. *For every fair execution of Procedure* ***ComputeLow****, if there is a suffix Π in which $path_i, 1 \le i \le n$, contain the correct values in every configuration, then there is a suffix of Π in which $path_i, low1_i$, $low2_i$, and $lowchild_i, 1 \le i \le n$, contain their correct values in every configuration.*

Proof: After the execution reaches the first configuration of Π, since $path_i, 1 \le i \le n$, contain the correct values, therefore at every leaf node v_i of T_{dfs}, after reading in the *path* values from all the outgoing non-tree links, $low1_i, low2_i$ are correctly computed. Moreover, $lowchild_i$ is also correctly set to *null* as the *to-low* link is a non-tree link. Therefore, there is a suffix of Π in which $path_i, low1_i, low2_i$ and $lowchild_i$, contain the correct values at every leaf-node v_i of T_{dfs}. Suppose there is a suffix Π' of Π in which in every configuration, $low1_j, low2_j, 1 \le j \le n$, contain the correct

```
ComputeLow(v_i, v_ij, 1 ≤ j ≤ δ_i):     /* Procedure for computing low1_i, low2_i, i ≥ 2 */
begin
    low1 := low2 := path := read(path_i);          /* initialize low1, low2, and path */
    lowchild := null;                                          /* initialize lowchild */
    for j := 1 to δ_i do
        if (readpath_j = path ⊕ j) then                    /* (v_i, v_ij) is a child link */
            readlow1_j := read(low1_ij) ;                  /* read low1 value of child v_ij */
            if (readlow1_j ≺ low1) then                          /* update low1, lowchild */
                low2 := low1; low1 := readlow1_j; lowchild := readpath_j;      /* and low2 */
            else low2 := min_≺(low2, readlow1_j);                          /* update low2 */
        else if (∃s)((path = readpath_j ⊕ s) ∧ (s ≠ α_ij(i))) then /* the link (v_i, v_ij) is
        an outgoing non-tree edge */
            if (readpath_j ≺ low1) then                          /* update low1, lowchild */
                low2 := low1; low1 := readpath_j; lowchild := null;            /* and low2 */
            else low2 := min_≺(low2, readpath_j);          /* update low2 accordingly */
    end
    write low1_i := low1; write low2_i := low2; write lowchild_i := lowchild;
end
```

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

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 $path_j, 1 \leq j \leq n$, $low1_i, low2_i$ and $lowchild_i$ are correctly determined. Hence, there is a suffix of the execution in which, for every node v_i, $1 \leq i \leq n$, $path_i$, $low1_i$, $low2_i$, 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*.

Initially, at any node v_i, the $low2_i$ and $path_i$ values are used to initialize the local variables *next*

```
 1 ComputeNestedPath(v_i, v_ij, 1 ≤ j ≤ δ_i):               /* Computing nestedpath_i, i ≥ 2 */
 2 begin
 3     next := low2:= read (low2_i);  nestedpath := path:= read (path_i);
 4     lowchild:= read (lowchild_i);
 5     if (lowchild ≠ null) then
 6         Let lowchild = path_ic be the lowchild;
 7         nestedpath := read (nestedpath_ic);          /* read nestedpath_ic from lowchild */
 8         Let the pairs of path values marked in nestedpath be (x_1, q_1), (x_2, q_2), ··· , (x_k, q_k) in
           nested order where for any pair (x_l, q_l), 1 ≤ l ≤ k, q_l and x_l are the path values up to
           the l-th (from start) $ symbol and up to l-th (from end) $ symbol, respectively, in
           nestedpath;
 9         if (readpath_c = q_k) then                                  /* unmark innermost pair */
10             nestedpath := unmark(nestedpath, (x_k, q_k));
           /* Now unmark all pairs interlaced with (v_i, low2_i)                                */
11         for every (x_l, q_l), 1 ≤ l ≤ k, marked in nestedpath such that low2 ⪯ q_l ⪯ path do
12             nestedpath := unmark(nestedpath, (x_l, q_l));
13         end
14         for j := 1 to δ_i do
               /* If the link (v_ij, v_i) is an incoming non-tree edge, then unmark all
               pairs interlaced with (v_ij, v_i)                                                 */
15             if (∃s)((readpath_j = path ⊕ s) ∧ (s ≠ α_i(i_j))) then
16                 for every (x_l, q_l), 1 ≤ l ≤ k, in nestedpath such that path ⪯ x_l ⪯ readpath_j do
17                     nestedpath := unmark(nestedpath, (x_l, q_l));
18                     next =min_≺(q_l, next);                              /* update next */
19                 end
20         end
       /* Insert innermost pair in nestedpath and write in register                            */
21     write nestedpath_i := mark(nestedpath, (path, next));  write next_i := next;
22 end
```

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

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 $nestedpath_{ic}$ is read into the local variable *nestedpath* (line 7). Let the sequence in *nestedpath* be $(x_1, q_1), (x_2, q_2), ..., (x_k, q_k)$, $k \geq 0$. If $path_{ic} = q_k$, 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 (x_l, q_l) that interlaces with the pair $(v_i, low2_i)$ (i.e. $low2_i \preceq q_l \preceq path_i$) is unmarked in *nestedpath* (lines 12-13) as the node q_l cannot be a representative node owing to Theorem 4.3. Similarly, for each incoming non-tree link, (v_{ij}, v_i), of v_i, every node-pair (x_l, q_l) interlacing with (v_{ij}, v_i) (i.e. $path_i \preceq x_l \preceq path_{ij}$) is unmarked in *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 $next_i$ accordingly (line 18). Finally, on line 21, $path_i$ and $next_i$ are marked in *nestedpath*; the latter is then recorded into the field $nestedpath_i$.

Lemma 6.3. *For every fair execution of Procedure **ComputeNestPath**, if there is a suffix Π in which $path_i$, $low1_i, low2_i$, and $lowchild_i, 1 \leq i \leq n$, contain the correct values in every configuration, then there is a suffix of Π in which, in every configuration, $nestedpath_i$ contains the correct representation of $S_{v_i}, 1 \leq i \leq n$, and $next_i$ contains the correct value.*

Proof: After the execution reaches the first configuration of Π, since $path_i, low1_i, low2_i$, and $lowchild_i$, $1 \leq i \leq n$, contain the correct values, at every leaf node v_i of T_{dfs}, the local variables $next, nestedpath$, and $lowchild$ are correctly initialized to $low2_i, path_i$, and $lowchild_i$, respectively. Furthermore, as v_i is a leaf-node, $lowchild$ must be $null$. As a result, $path_i$ and $next_i$ are correctly marked in *nestedpath* (which is $path_i$) and $next_i$ is correctly set to $low2_i$ on Line 21. Therefore, there is a suffix of Π in which $nestedpath_i$ and $next_i$ contain the correct values at every leaf-node v_i of T_{dfs}.

Suppose there is a suffix Π' of Π in which, in every configuration, the fields $path_j, low1_j, low2_j$, $lowchild_j$, and $nestedpath_j$, $1 \leq j \leq n$, 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 *lowchild* of v_i is *null*, then $path_i$ and $next_i$ are marked in *nestedpath* (which is $path_i$) which correctly represents the sequence $S_{v_i} : (v_i, low2(v_i))$ and $next_i$ is correctly set to $low2_i$ on Line 21. Otherwise, node v_i reads $nestedpath_c$ from the *lowchild* v_{i_c} and stores the value in the local variable *nestedpath*. Since v_{i_c} is on level h, *nestedpath* thus contains the correct representation of $S_{v_{i_c}}$. Node v_i then updates *nestedpath* by unmarking (removing) all those (x_l, q_l) pairs that interlace with $(v_i, low2_i)$ or with some incoming non-tree link, (v_{i_j}, v_i), of v_i, and adding $(v_i, next_i)$ to *nestedpath* as the innermost pair. Node v_i also updates the local variable *next* to the smallest (w.r.t. $\prec$) $path_{q_l}$ of those (x_l, q_l)'s that are unmarked, if $path_{q_l} \prec low2_i$ for some l. Finally, the correct values of *nestedpath* and *next* are written into the fields $nestedpath_i$ and $next_i$, respectively.

Hence, there is a suffix of the execution in which, for every node v_i, $1 \leq i \leq n$, $nestedpath_i$ and $next_i$ contain their correct values. □

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 $reprenode(v_i)$ 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 $next_i$ is a proper prefix of tcc_j (i.e. $next_i \prec tcc_j$).*

Proof: Suppose $next_i \not\preceq tcc_j$. 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. □

The above lemma shows that for every non-representative node v_i, if $next_i \not\prec tcc_j$, 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 $path_j$ is a prefix of $path_i$, as with *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 $path_i$ 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 $path_j$ (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. ***remove\$***(*rtcc*): Returns the *path* value after removing all the \$ symbols from *rtcc*.

   For example, ***remove\$***$(\alpha_1\$\alpha_2\alpha_3\$) = \alpha_1\alpha_2\alpha_3$.

2. ***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,

delnode$(\alpha_1\$\alpha_2\alpha_3\$\alpha_4\alpha_5\$\alpha_6\$) = \alpha_1\$\alpha_2\alpha_3\$\alpha_4\alpha_5\$$; ***delnode***$(\alpha_1\$\alpha_2\alpha_3\$\alpha_4\alpha_5\alpha_6\$) = \alpha_1\$\alpha_2\alpha_3\$$; ***delnode***$(\alpha_1\alpha_2\$) = \perp$.

3. ***track***$(path, rtcc)$: Let suf be the suffix of $path$ such that ***remove\$***$(rtcc) \oplus suf = path$. Then ***track***$(path, rtcc)$ returns $rtcc \oplus suf \oplus \$$. For example, ***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 ≤ j ≤ δ_i):                /* Procedure for computing tcc_i, i ≥ 2 */
2  begin
3     next := read(next_i); path :=read(path_i); nestedpath :=read(nestedpath_i);
4     for j := 1 to δ_i do
5        if (path = readpath_j ⊕ α_ij(i)) then              /* (v_ij, v_i) is the parent link */
6           readrtcc_j :=read(rtcc_ij); readtcc_j =remove$(readrtcc_j);      /* read tcc_ij */
7           if (path = next) then                        /* v_i is a representative node */
8              tcc := path;                          /* copy its path value into its tcc */
9              if (∃(x_h, q_h) in nestedpath such that path_{q_h} = readtcc_j) then
10                rtcc :=track(path, readrtcc_j)
11             else
12                rtcc :=track(path, delnode(readrtcc_j));                /* remove tcc_j */
13          else                                     /* v_i is not a representative node */
14             if (readtcc_j ⪯ next) then                  /* v_i, v_ij are of same component */
15                tcc := readtcc_j; rtcc := readrtcc_j;        /* tcc of v_i, v_ij are same */
16             else /* v_i, v_ij are of different component                             */
17                rtcc :=delnode(readrtcc_j); tcc :=remove$(rtcc);  /* extract tcc from
                  tcc_ij */
18       end
19    end
20    write tcc_i := tcc; write rtcc_i := rtcc;       /* write tcc and rtcc into register */
21 end
```

Procedure `ComputeID`(v_i, v_{ij}, $1 \le j \le \delta_i$)

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

If $next_i = path_i$ (Line 7), then v_i is a representative node by Theorem 5.4. So, node v_i stores $path_i$ in tcc_i (Line 8). Furthermore, if tcc_{i_j} appears in a node-pair in $nestedpath_i$ (Line 9), then it indicates that tcc_{i_j} 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_{i_j} 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 $tcc_{i_j} \preceq next_i$, then by Lemma 6.4, v_i and v_{i_j} belong to the same 3-edge-connected component. So, node v_i simply copies tcc_{i_j} and $rtcc_{i_j}$ into tcc_i and $rtcc_i$, respectively (Line 15). Otherwise, v_i and v_{i_j} belong to different 3-edge-connected components. Since v_{i_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 $rtcc_i$, $rtcc_i$, 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* Π *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* Π *in which, in every configuration,* $tcc_i = reprenode(v_i), 1 \leq i \leq n$.

Proof: Suppose the execution has reached a configuration in Π. 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* Π *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* $q_i, 1 \leq i < k$, *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* T_{dfs}.

The root v_1 is the only node on level 0. Since v_1 always keeps the constant $\perp$ and $\perp\$$ 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 $rtcc_{i_j}$ from its parent node, v_{i_j}, which is on level h, by assumption, both $rtcc_{i_j}$ and tcc_{i_j} 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 $path_{q_m} = tcc_{i_j}$, then tcc_{i_j} 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 $rtcc_{i_j}$. Since $rtcc_{i_j}$ 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_{i_j} is not 3-edge-connected to any descendant of v_i. Nodes v_i thus correctly executes Line 12 which removes tcc_{i_j} from $rtcc_{i_j}$ before adding tcc_i to $rtcc_{i_j}$. Again, the resulting $rtcc_i$ clearly satisfies the condition given in the assertion.

1 **root** v_1:
2 **for** *forever* **do**
3   **write** $path_1 := \perp$; **write** $tcc_1 := \perp$; **write** $rtcc_1 := \perp\$$;
4 **end**
5 **non-root** v_i, $i \geq 2$:
6 Let v_{i_j}, $1 \leq j \leq \delta_i$, be the neighboring processors of processor v_i, $2 \leq i \leq n$, such that $\alpha_i(i_j) = j$, $1 \leq j \leq \delta_i$, $2 \leq i \leq n$.
7 **for** *forever* **do**
8   **ComputePath**(v_i, v_{i_j}, $1 \leq j \leq \delta_i$); `/* Compute` $path_i$ `*/`
9   **ComputeLow**(v_i, v_{i_j}, $1 \leq j \leq \delta_i$); `/* Compute` $low1_i, low2_i$ `*/`
10   **ComputeNestedPath**(v_i, v_{i_j}, $1 \leq j \leq \delta_i$); `/* Compute` $nestedpath_i$ `*/`
11   **ComputeID**(v_i, v_{i_j}, $1 \leq j \leq \delta_i$); `/* Compute` tcc_i `*/`
12 **end**

Algorithm 5: 3-EDGE-CONNECTIVITY

If $path_i \neq next_i$, then by Theorem 5.4, v_i is not a representative node. Furthermore, if $tcc_{i_j} \preceq next_i$, then by Lemma 6.4, v_i and v_{i_j} belong to the same 3-edge-connected component. Node v_i thus correctly copies tcc_{i_j} and $rtcc_{i_j}$ into tcc_i and $rtcc_i$, respectively on Line 15. The assertion clearly holds true for node v_i. On the other hand, if $next_i \prec tcc_{i_j}$, then v_i and v_{i_j} 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_{i_j}. As a result, node v_i correctly removes tcc_{i_j} from $rtcc_{i_j}$ 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 ***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, $tcc_i = reprenode(v_i), 1 \leq i \leq n$.

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 **ComputePath**, **ComputeLow**, **ComputeNested-Path**, and **ComputeID** in that order for computing $path_i$, $low1_i$ and $low2_i$, $nestedpath_i$, tcc_i, respectively. In each procedure, processor v_i reads from its neighboring processors v_{i_j}, $1 \leq j \leq \delta_i$, and writes into its own register. When the algorithm stabilizes, the *path* value of $reprenode(v_i)$ is kept in tcc_i.

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 Δ 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 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 **3-EDGE-CONNECTIVITY** algorithm for an undirected graph whose nodes are $v_1, v_2, \cdots, v_{11}$. For every node v_i, $2 \leq i \leq 11$,

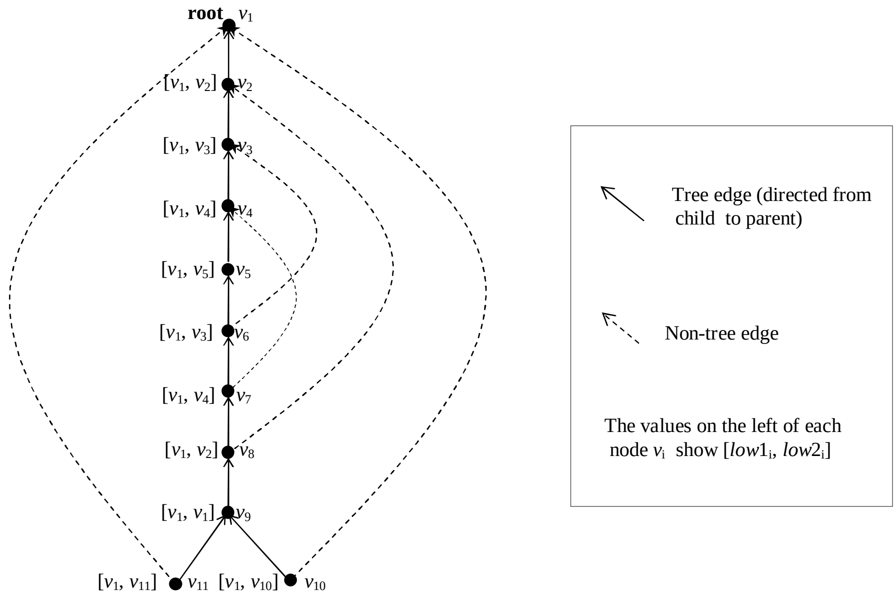

Figure 3: A Depth-First Search Spanning Tree T_{dfs}. In this graph, cut-pairs are: $\{(v_9, v_{10}), (v_{10}, v_1)\}; \{(v_9, v_{11}), (v_{11}, v_1)\}; \{(v_8, v_9), (v_1, v_2)\}; \{(v_7, v_8), (v_2, v_3)\}; \{(v_5, v_6), (v_4, v_5)\}$ and 3-edge-connected components are: $\{v_{10}\}; \{v_{11}\}; \{v_9, v_1\}; \{v_8, v_2\}; \{v_7, v_6, v_4, v_3\}; \{v_5\}$

the figure also shows the values of $low1_i$ and $low2_i$ 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: $\{(v_{11}, v_{11})\}$, $\{(v_{10}, v_{10})\}$, $\{(v_9, v_1)\}$, $\{(v_9, v_1), (v_8, v_2)\}$, $\{(v_9, v_1), (v_8, v_2), (v_7, v_4)\}$, $\{(v_9, v_1), (v_8, v_2), (v_6, v_3)\}$, $\{(v_9, v_1), (v_8, v_2), (v_6, v_3), (v_5, v_5)\}$, $\{(v_9, v_1), (v_8, v_2), (v_4, v_3)\}$, $\{(v_9, v_1), (v_8, v_2), (v_3, v_3)\}$, and $\{(v_9, v_1), (v_2, v_2)\}$, respectively. When the algorithm stabilizes, $tcc_1 = tcc_9 = path_1$; $tcc_{10} = path_{10}$; $tcc_{11} = path_{11}$; $tcc_5 = path_5$; $tcc_2 = tcc_8 = path_2$; $tcc_3 = tcc_4 = tcc_6 = tcc_7 = path_3$. Therefore, the 3-edge-connected components are: $\{v_1, v_9\}$, $\{v_{10}\}$, $\{v_{11}\}$, $\{v_5\}$, $\{v_2, v_8\}$, and $\{v_3, v_4, v_6, v_7\}$.

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

[1] Antonoiu, G., and Srimani, P. K. A self-stabilizing distributed algorithm to find the median of a tree graph. *Journal of Computer and System Science 58*, 1 (February 1999), 215–221.

[2] Chaudhuri, P. A self-stabilizing algorithm for detecting fundamental cycles in a graph. *Journal of Computer and System Science 59*, 1 (August 1999), 84–93.

[3] Chaudhuri, P. An $\mathbf{O}(n^2)$ self-stabilizing algorithm for computing bridge-connected components. *Computing 62*, 1 (February 1999), 55–67.

[4] Collin, Z., and Dolev, S. Self-stabilizing depth-first search. *Information Processing Letters 49*, 6 (March 1994), 297–301.

[5] Devismes, S. A silent self-stabilizing algorithm for finding cut-nodes and bridges. *Parallel Processing Letters 15*, 1&2 (March & June 2005), 183–198.

[6] Dijkstra, E. W. Self-stabilizing systems in spite of distributed control. *Communications of the ACM 17*, 1 (November 1974), 643–644.

[7] Dijkstra, E. W. A belated proof of self-stabilization. *Distributed Computing 1*, 1 (January 1986), 5–6.

[8] Dolev, S. *Self-stabilization.* MIT Press, Cambridge, Massachusetts, 2000.

[9] Even, S. *Graph Algorithms.* Computer Science Press, Potomac, Maryland, 1979.

[10] Ghosh, S., and Karaata, M. H. A self-stabilizing algorithm for coloring planar graphs. *Distributed Computing 7*, 1 (1993), 55–59.

[11] Gradinariu, and Tixeuil, S. Self-stabilizing vertex coloring of arbitrary graphs. In *International Conference on Principles of Distributed Systems (OPODIS'2000) in Paris, France* (2000).

[12] Karaata, M. H. A self-stabilizing algorithm for finding articulation points. *International Journal of Foundations of Computer Sciences 10*, 1 (1999), 33–46.

[13] karaata, M. H. A stabilizing algorithm for finding biconnected components. *Journal of Parallel and Distributed Computing 62*, 5 (May 2002), 982–999.

[14] Karaata, M. H., and Chaudhuri, P. A self-stabilizing algorithm for bridge finding. *Distributed Computing 12*, 1 (March 1999), 47–53.

[15] Lin, J.-C., and Huang, T. C. An efficient fault-containing self-stabilizing algorithm for finding a maximal independent set. In *IEEE Transactions on Parallel and Distributed Systems* (August 2003), vol. 14, pp. 742–754.

[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] Saifullah, A. M., and Tsin, Y. H. A self-stabilizing algorithm for 3-edge-connectivity. In *Parallel and Distributed Processing and Applications, 5th International Symposium, ISPA 2007 (LNCS 4742), Niagara Falls, Canada.* (August 2007). Also to appear in a special issue of the International Journal of High Performance Computing and Networking (IJHPCN).

[18] Tarjan, R. E. Depth-first search and linear graph algorithms. *SIAM J. Computing 1* (1972), 146–160.

[19] Tsin, Y. H. An efficient distributed algorithm for 3-edge-connectivity. *International Journal of Foundations of Computer Science 17*, 3 (2006), 677–701.

[20] Tsin, Y. H. An improved self-stabilizing algorithm for biconnectivity and bridge-connectivity. *Information Processing Letters 102* (2007), 27–34.