Report Number: WUCSE-2011-96

2011

2-Edge-Connectivity and 2-Vertex-Connectivity with Fault Containment

Abusayeed Saifullah

Self-stabilization for non-masking fault-tolerant distributed system has received considerable research interest over the last decade. In this paper, we propose a self-stabilizing algorithm for 2-edge-connectivity and 2-vertex-connectivity of an asynchronous distributed computer network. It is based on a self-stabilizing depth-first search, and is not a composite algorithm in the sense that it is not composed of a number of self-stabilizing algorithms that run concurrently. The time and space complexities of the algorithm are the same as those of the underlying self-stabilizing depth-first search algorithm.

... Read complete abstract on page 2.

2-Edge-Connectivity and 2-Vertex-Connectivity with Fault Containment

Abusayeed Saifullah

Complete Abstract:

Self-stabilization for non-masking fault-tolerant distributed system has received considerable research interest over the last decade. In this paper, we propose a self-stabilizing algorithm for 2-edge-connectivity and 2-vertex-connectivity of an asynchronous distributed computer network. It is based on a self-stabilizing depth-first search, and is not a composite algorithm in the sense that it is not composed of a number of self-stabilizing algorithms that run concurrently. The time and space complexities of the algorithm are the same as those of the underlying self-stabilizing depth-first search algorithm.

2011-96

2-Edge-Connectivity and 2-Vertex-Connectivity with Fault Containment

Authors: Abusayeed Saifullah

Corresponding Author: saifullaha@cse.wustl.edu

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

Abstract: Self-stabilization for non-masking fault-tolerant distributed system has received considerable research interest over the last decade. In this paper, we propose a self-stabilizing algorithm for 2-edge-connectivity and 2-vertex-connectivity of an asynchronous distributed computer network. It is based on a self-stabilizing depth-first search, and is not a composite algorithm in the sense that it is not composed of a number of self-stabilizing algorithms that run concurrently. The time and space complexities of the algorithm are the same as those of the underlying self-stabilizing depth-first search algorithm.

Type of Report: Other

2-Edge-Connectivity and 2-Vertex-Connectivity with Fault Containment

Abusayeed Saifullah
Computer Science and Engineering
Washington University in St Louis
saifullaha@cse.wustl.edu

ABSTRACT
Self-stabilization for non-masking fault-tolerant distributed system has received considerable research interest over the last decade. In this paper, we propose a self-stabilizing algorithm for 2-edge-connectivity and 2-vertex-connectivity of an asynchronous distributed computer network. It is based on a self-stabilizing depth-first search, and is not a composite algorithm in the sense that it is not composed of a number of self-stabilizing algorithms that run concurrently. The time and space complexities of the algorithm are the same as those of the underlying self-stabilizing depth-first search algorithm which are $O(dn\Delta)$ rounds and $O(n \log \Delta)$ bits per processor, respectively, where $\Delta(\leq n)$ is an upper bound on the degree of a node, $d(\leq n)$ is the diameter of the graph, and n is the number of nodes in the network.

KEY WORDS
Distributed system, fault-tolerance, self-stabilization, depth-first search tree, bridge, articulation point, bridge-connected component.

1 Introduction

A *distributed system* is a set of processing elements or state machines interconnected by a network of some fixed topology. Distributed systems are exposed to constant changes of their environment and the design of such systems is quite complex, in part due to unpredictable faults. Implicit in the notion of *fault* is the specification of what constitutes the correct state of the system. A *transient fault* is an event that may change the state of a system by corrupting the local states of the machines. The property of *self-stabilization* can recover the system from transient faults and represents a departure from previous approaches to fault tolerance.

The notion of *self-stabilization* was first proposed by Dijkstra [5, 6]. A system is *self-stabilizing* if, starting at any state, possibly illegitimate, it eventually converges to a legitimate state in finite time. A self-stabilizing system is capable of tolerating any unexpected transient fault without being assisted by any external agent. Regardless of the initial state, it can reach a legitimate global state in finite time and can remain so thereafter unless it experiences any subsequent fault. In this paper, we propose a simple self-stabilizing algorithm for detecting the bridges, articulation points, and bridge-connected components of an asynchronous distributed network. When a distributed system is modelled as an undirected connected graph, an edge is called a *bridge* if its removal disconnects the graph whereas an *articulation point* is a node whose removal disconnects the graph. A maximal component without any bridge of the graph is called a *bridge-connected component*. Bridge-connectivity (2-edge-connectivity) and biconnectivity (2-vertex-connectivity) call for considerable attention in graph theory since these properties represent the extent to which a graph is connected. In distributed systems, these properties represent the reliability of the network in presence of link or node failures. Moreover, when communication links are expensive, these properties play a vital role to minimize the communication cost.

Several self-stabilizing algorithms for 2-edge-connectivity and 2-vertex-connectivity are available. The algorithm in [1] can find the bridge-connected components by assuming the existence of a depth-first search spanning tree of the system. This algorithm stabilizes in two phases and, for a system with n processors, each phase requires $O(n^2)$ moves to reach a legitimate configuration by assuming that the preceding phase has stabilized. If a breadth-first search tree of the network is known, then the algorithm in [10] can detect the bridges in $O(n^3)$ moves and that in [8] can detect the articulation points in $O(n^3)$ moves. The algorithm in [9] finds the biconnected components in $O(n^2)$ moves if a breadth-first search tree and all the bridges of the network are known. Each of the algorithms [1, 8, 9, 10] mentioned above requires $O(n\Delta \lg \Delta)$ bits per processor, where Δ is an upper bound on the *degree* of a processor. The algorithm proposed by Devismes [4] uses a weaker model (one that does not require every node to have a distinct identifier) and can detect the cut-nodes and bridges in $O(n^2)$ moves if a depth-first search tree of the network is known. This algorithm is memory efficient ($O(n \lg \Delta + \lg n)$ bits per processor) but does not find the bridge-connected or biconnected components.

It is pointed out in [12] that each of the aforementioned algorithms is just one component of a composite algorithm and hence the time complexity presented is different from that of the composite algorithm. Since the algorithm must run concurrently with a self-stabilizing spanning tree algorithm (which is another component of the composite algorithm), when the last transient fault had elapsed and the spanning tree algorithm has stabilized, the processor may make redundant moves on the spanning tree algorithm which could significantly lengthen the time that the composite algorithm needs to stabilize. In the worst case, the time complexity of the composite algorithm is the $product$ of the time complexities of the algorithms that make up the composite algorithm and is thus bounded below by that of the spanning tree algorithm. Addressing all these issues, Tsin [12] has shown how to incorporate Tarjan's depth-first-search based algorithm for biconnectivity into the self-stabilizing depth-first search algorithm of Collin and Dolev [3] to produce a self-stabilizing algorithm for bridge-connectivity and biconnectivity. The time and space complexities of the resulting algorithm are bounded above by those of the depth-first search algorithm. Following this elegant approach [12], our algorithm simplifies all existing algorithms for bridge-connectivity and biconnectivity [1, 4, 8, 9, 10] by embedding the detection method of bridges and articulation points in the self-stabilizing depth-first search algorithm of Collin and Dolev [3] and by avoiding any distributed protocol composition. The proposed algorithm also determines all the bridge-connected components since, upon stabilization of the algorithm, all the nodes of the same component contain the same identifier. The space complexity is also significantly improved in our algorithm. The space requirement for each of the algorithms of [1, 8, 9, 10] is $O(n^2 \log(n))$ bits per processor for a system with n processors. This is due to the propagation of a set of non-tree edges that bypass a tree edge in the depth-first search spanning tree of the system. However, we show that passing only the size of that set is sufficient for detecting all the bridges and articulation points which substantially reduces the size of the message. Specifically, the time complexity of our algorithm is $O(dn\Delta)$ rounds and the space complexity for every processor is $O(n \log \Delta)$ bits. Note that the space complexity for the self-stabilizing depth-first algorithm of Collin and Dolev [3] is $O(n \log \Delta)$ bits per processor and the time complexity is $O(dn\Delta)$ rounds. The model we use is the same as that of Collin and Dolev [3], which is weaker than that used in [1, 2, 8, 9, 10].

2 Computational Model

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* between two processors. We shall use the terms *node* and *processor* (*edge* and *link*, respectively) interchangeably throughout this paper. 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 $i \geq 0$, $c_i \rightarrow c_{i+1}$ (called a ***single computation step***) denotes that configuration c_{i+1} can be reached from configuration c_i by executing on 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 the number of ***rounds*** [7]. 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 $\Re_2$.

3 The Algorithm

The algorithm uses the self-stabilizing depth-first search algorithm of Collin and Dolev [3] to construct a depth-first search spanning tree. In the self-stabilizing depth-first search algorithm of Collin and Dolev [3], 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 and the *concatenation* of any link with a path is denoted by the operator $\oplus$. The root processor v_1 always writes $\perp$ in its $path_1$

field and, in the lexicographical order relation, $\perp$ is the *minimal character*. When a depth-first search tree is constructed in the network, $path_i$ contains the smallest (with respect to the lexicographical order $\prec$) path connecting v_1 with v_i. The last links on the smallest paths of v_i, $i \geq 2$, form a depth-first search tree, called the *first depth-first search tree*. Given that in the first depth-first search tree, a node v_j is an ancestor of a node v_i if the smallest path of v_i contains the smallest path of v_j, then, the node v_j is an ancestor of a node v_i if $path_j$ is a prefix of $path_i$, i.e. $(\exists s)(path_i = path_j \oplus s)$. If there exists a unique neighbor v_j of v_i such that $path_i = path_j \oplus \alpha_j(i)$, then v_j is the *parent* of v_i. The ***degree of a processor*** v_i, denoted by δ_i, is the number of incident edges (links) on v_i. Once a depth-first search tree is constructed, at each processor 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 $\alpha_j(i)$ in the following ways:

- The link (v_j, v_i) is a ***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*** (i.e. it is a non-tree link and v_j is an ancestor of v_i) if and only if $(\exists s)((path_i = path_j \oplus s) \wedge (s \neq \alpha_j(i)))$; The total number of outgoing non-tree edges incident on processor v_i is denoted by out_i.
- The link (v_j, v_i) is an ***incoming non-tree edge*** (i.e. it is a non-tree link and v_j is a descendant of v_i) if and only if $(\exists s)((path_j = path_i \oplus s) \wedge (s \neq \alpha_i(j)))$; The total number of incoming non-tree edges incident on processor v_i is denoted by in_i.

We omit the description of that part of the algorithm for constructing a depth-first search tree T, as it is available in [3]. The idea underlying our algorithm is to count the total number of non-tree edges that bypass a tree edge in T. A non-tree edge (v_k, v_l) (v_k is a descendant of v_l) ***bypasses*** a tree edge (v_i, v_j) (v_i is the parent of v_j) if and only if v_k is a descendant of v_j while v_l is an ancestor of v_i. The total number of non-tree edges bypassing the parent link of processor v_i is denoted by $count_i$. During the execution of the algorithm this number is propagated towards the root whereas in [1, 10, 8, 9], for every node v_i, the whole set of non-tree edges bypassing the parent link of v_i is calculated and routed towards the root. Since, in our algorithm, only the cardinality of the set is propagated, the message cost is drastically reduced. IN T, let in_i and out_i be the number of incoming non-tree edges and the number of outgoing non-tree edges, respectively, incident on v_i, and C_i be the set of children of v_i, and $incoming(v_j, v_i)$ be the number of incoming non-tree edges (v_l, v_i) such that v_l is a descendant of $v_j \in C_i$. Then $count_i$ is calculated recursively as follows:

$$count_i := \sum_{v_j \in C_i} count_j - in_i + out_i;$$

The algorithm is based on the Theorem 1 and Theorem 2 due to Tarjan [11].

Theorem 1. *(i) If a non-root node v_i has a child v_j in T, then v_i is an articulation point of G if and only if $count_j = incoming(v_j, v_i)$.*

(ii) The root v_1 is an articulation point of G if and only if v_1 has two or more children.

Theorem 2. *Let (v_i, v_j) be a tree edge in T such that v_i is the parent of v_j. Then (v_i, v_j) is a bridge in G if and only if $count_j = 0$.*

Corollary 1 follows from Theorem 1 and Theorem 2.

Corollary 1. *Each of the end nodes of a bridge is an articulation point unless it is a node of degree one.*

Remark 1. *For each leaf node v_i, $in_i = 0$ and $count_i = out_i$.*

In order to extend this depth-first search algorithm to find the bridges and articulation points, and bridge-connected components every processor v_i, in addition to the field $path_i$, maintains two fields: $count_i$ and bcc_i. The field bcc_i is a unique identifier of the bridge-connected component containing v_i. For every bridge-connected component, a ***representative node*** is defined. A ***representative node*** v_j of a bridge-connected component is the ancestor of all other nodes of the component containing v_j. When the algorithm stabilizes, every bridge-connected component is uniquely identified by the $path$-value of its *representative node*, and bcc-fields of all nodes of this component contain this $path$-value.

Lemma 1. *A node v_i is a representative node if and only if $count_i = 0$.*

Proof. Let v_i be a *representative node* and $count_i > 0$. Let (v_m, v_l) be a non-tree edge such that v_l is an ancestor and v_m is a descendant of v_i. Node v_l can be reached from v_i using the tree path $v_i - v_m$ followed by the non-tree edge (v_m, v_l) while v_l can also be reached from v_i using another path $v_i - v_l$ and these two paths are disjoint. That is, the ancestor v_l is bridge-connected to v_i which contradicts that v_i is a *representative node*. Again, by Theorem 2, if $count_i = 0$ then no ancestor of v_i can be reached from v_i when the parent link of v_i is removed. Hence v_i is a *representative node* of the bridge-connected component containing v_i, □

During the execution of the algorithm, every non-root node $v_i, i \geq 2$, repeatedly reads in $count_j$ of every $v_j \in C_i$, and based on in_i and out_i, it counts the value $count_i$. Furthermore, every *representative node* v_i repeatedly writes its own path value $path_i$ into bcc_i field and every *non-representative node* v_l repreatedly reads in bcc_m of its parent v_m and writes this value into bcc_l. The root v_1 always writes 0 into $count_1$ field and $\perp$ (i.e. $path_1$ value) into bcc_1 field.

The algorithm is presented as the **2-EDGE & 2-VERTEX CONNECTIVITY** algorithm. The functions ***read*** and ***write*** are the functions for reading from and writing to a register, respectively. The fields in the register of v_i, $1 \leq i \leq n$, are: $path_i$, $count_i$, bcc_i; the local variables are $path$, $read\_path_j$, $read\_count_j$, and $read\_bcc_j$ $(1 \leq j \leq \delta_j)$.

Theorem 3. *For every fair execution of the **2-EDGE & 2-VERTEX CONNECTIVITY** algorithm, there is a suffix in which for every node v_i, $1 \leq i \leq n$, $bcc_i = path_t$ in every configuration, where v_t, $1 \leq t \leq n$, is the **representative node** of the bridge-connected component containing v_i.*

Proof. In the **2-EDGE & 2-VERTEX CONNECTIVITY** algorithm, new instructions for determining the bridges, and articulation points are embedded in the self-stabilizing depth-first search algorithm of Collin and Dolev [3]. These new instructions do not affect the original function of the depth-first search algorithm. Therefore, by Theorem 3.2 in [3], for every fair execution of the **3-EDGE-CONNECTIVITY** algorithm, there is a suffix S of the execution in which $path_i$, $1 \leq i \leq n$, contains the correct value in every configuration. Suppose the execution has reached a configuration c in S. By Observation 3.1 in [3], the correct values in $path_i$, $1 \leq i \leq n$, specify a depth-first search tree T.

Let v_i be any leaf node in T. Since $path_j$, $1 \leq i \leq n$, is correctly determined, after v_i reads in the $path$ field from each outgoing non-tree link, $count_i$ value is correctly determined. Let S' be a suffix of the execution in which all the nodes on level h or higher (i.e. farther from the root) have correctly computed their $count$ values. Consider any non-leaf node v_i, on level $h - 1$. By the induction hypothesis, for each $v_j \in C_i$, the values of $count_j$ are correctly calculated. Therefore, v_i correctly calculates $count_i$. Hence, there is a suffix of the suffix S' in which for every configuration, $count_i$ are correctly computed for every node v_i, $1 \leq i \leq n$.

Suppose the execution has reached a configuration in the aforementioned suffix of suffix S'. The root node v_1 is a *representative node* and always correctly writes the value of $path_1$ (i.e. $\perp$) into the field bcc_1. Let S'' be a suffix in the suffix S' of the execution in which, for every node v_m on level h or lower (i.e.

closer to the root), $bcc_m = path_l$, where v_l is the *representative* node of the bridge-connected component containing v_m. Let v_i be any non-root node on level $h+1$. The value $next_i$ can be read from S_i which is correctly calculated. If v_i is a *representative* node, then v_i correctly writes $path_i$ into bcc_i. If v_i is not a *representative* node, then v_i reads in the bcc-field of its parent which is correct by the induction hypothesis and writes it into bcc_i. Hence, there is a suffix of the fair execution in which for every node v_i, $1 \leq i \leq n$, $bcc_i = path_t$ in every configuration, where v_t, $1 \leq t \leq n$, is the *representative node* of the bridge-connected component containing v_i. □

2-EDGE & 2-VERTEX CONNECTIVITY Algorithm

Let v_{i_j}, $1 \leq j \leq \delta_i$, be the neighboring processors of processor v_i, $1 \leq i \leq n$, such that $\alpha_i(i_j) = j$, $1 \leq j \leq \delta_i$, $1 \leq i \leq n$.

```
root v_1:
for forever do
    write path_1 := ⊥; write count_1 := 0; write bcc_1 := ⊥;
end
non-root v_i, i ≥ 2:
for forever do
    /* Calculate path_i                                                          */
    for j := 1 to δ_i do read_path_j := read(path_{i_j});  /* read path-value of neighbor v_{i_j}  */
    write path_i := min_≺{|read_path_j ⊕ α_{i_j}(i)|_N such that 1 ≤ j ≤ δ_i};   /* compute path_i  */
    /* Calculate count_i                                                         */
    path := read(path_i);
    in := out := count := 0;                      /* initialize in, out, and count  */
    for j := 1 to δ_i do
        if (read_path_j = path ⊕ α_{i_j}(i)) then              /* (v_i, v_{i_j}) is a child link */
            read_count_j := read(count_{i_j});             /* read count value of child v_{i_j} */
            count := count + read_count_j;                               /* update count */
        end
        else if (∃s)((read_path_j = path ⊕ s) ∧ (s ≠ α_{i_j}(i))) then /* incoming non-tree edge */
            count := count − 1;                                          /* update count */
        else if (∃s)((path = read_path_j ⊕ s) ∧ (s ≠ α_{i_j}(i))) then /* outgoing non-tree edge */
            count := count + 1;                                          /* update count */
    end
    write count_i := count;                                              /* write count_i  */
    /* Calculate bridge-connected component identifier bcc_i                     */
    count := read(count_i); path := read(path_i);
    if (count = 0) then
        write bcc_i := path; /* v_i is a representative node.  Write path_i into bcc_i  */
    else
        for j := 1 to δ_i do
            if (path = read_path_j ⊕ α_{i_j}(i)) then            /* ((v_{i_j}, v_i) is the parent link */
                read_bcc_j := read(bcc_{i_j}); write bcc_i := read_bcc_j; /* (write bcc of v_{i_j} in bcc_i */
            end
        end
end
```

Lemma 2. *When the **2-EDGE & 2-VERTEX CONNECTIVITY** algorithm stabilizes, all the bridges, articulation points, and bridge-connected components are determined.*

Proof. When the algorithm stabilizes, by Theorem 3, every node v_i, $1 \leq i \leq n$, knows its children, parent, all incident tree-edges and non-tree edges, $count_i$ values, and bcc_i values. By Theorem 2, any tree edge

(v_i, v_j) with $count_i = count_j = 0$ is a bridge, and, by Corollary 1, each of these two nodes is an articulation point unless its degree is one. By Theorem 1, any other non-root node v_i having a child C_j such that $count_j = incoming(v_j, v_i)$ is an articulation point. If the root v_1 has more than one child then v_1 is an articulation point. Every bridge-connected component has a unique *representative node* and, by Theorem 3, the *path* value of this node is written into *bcc* field of every node of this component. Hence bcc_i value of every node v_i, $1 \leq i \leq n$, uniquely identifies the bridge-connected component containing v_i. □

Lemma 3. *The* ***2-EDGE & 2-VERTEX CONNECTIVITY*** *algorithm stabilizes in O($dn\Delta$) rounds, where Δ is an upper bound on the degree of a node, d is the diameter of the graph.*

Proof. It is easily verified that the new instructions added to the depth-first search algorithm of Collin and Dolev [3] only increase the time complexity for constructing a depth-first search tree by a constant factor. The **for** loop for computing the *count*-values takes $O(H)$ rounds, where H is the height of T and the **for** loop for computing the *bcc* values takes O(1) rounds. Therefore, the time required by the **2-EDGE & 2-VERTEX CONNECTIVITY** algorithm remains same as that of the underlying depth-first search algorithm (i.e. O($dn\Delta$) rounds). □

Lemma 4. *The space complexity of the* ***2-EDGE & 2-VERTEX CONNECTIVITY*** *algorithm is O($n \log \Delta$) bits per processor.*

Proof. In the depth-first search algorithm of Collin and Dolev [3], 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 the **2-EDGE & 2-VERTEX CONNECTIVITY** algorithm, *bcc* field requires O($n \log \Delta$) bits, and *count* filed requires O($\log(n\Delta)$) $\approx$ O($\log n$) bits. The space complexity per processor is thus O($n \log \Delta$) bits. □

Figure 1 is a depth-first spanning tree of the corresponding undirected graph. An execution of our algorithm over this tree is shown below.

The *count* values at non-root nodes v_4, v_6, v_{11}, and v_{14} are 0, and hence, by Lemma 2, the bridges are (v_{11}, v_{14}), (v_{10}, v_{11}), (v_5, v_6), (v_1, v_4). The articulation points are v_1, v_4, v_5, v_6, v_{10}, v_{11}, v_{14}, and bridge-connected components are: $\{v_1, v_2, v_3\}$, $\{v_4, v_5, v_{10}\}$, $\{v_6, v_7, v_8, v_9\}$, $\{v_{11}, v_{12}, v_{13}\}$, and $\{v_{14}, v_{15}, v_{16}\}$.

4 Conclusion

We have presented an algorithm for the 2-edge-connectivity and 2-vertex-connectivity problem based on a self-stabilizing depth-first search algorithm. The algorithm constructs a depth-first search tree in $O(dn\Delta)$ rounds and then determines the bridges, articulation points, and bridges-connected components based on the depth-first search tree. In the worst case, when $d = \Delta = n$, our algorithm requires $O(n^3)$ rounds. Clearly, the time complexity of our algorithm is dominated by the time spent in constructing the depth-first search tree. Should there be an improvement made on the time bound required to construct the depth-first search tree, the time complexity of our algorithm will improve as well.

References

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

[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] COLLIN, Z., AND DOLEV, S. Self-stabilizing depth-first search. *Information Processing Letters 49*, 6 (March 1994), 297–301.

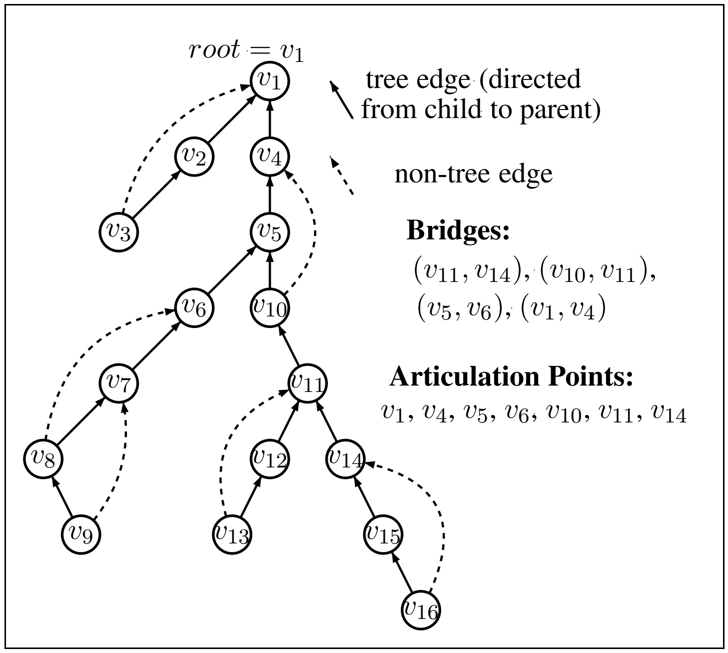

Figure 1: Depth-First Search Spanning Tree T

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

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

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

[7] DOLEV, S. *Self-stabilization*. MIT Press, Cambridge, Massachusetts, 2000.

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

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

[10] KARAATA, M. H., AND CHAUDHURI, P. A self-stabilizing algorithm for bridge finding. *Distributed Computing 12*, 1 (March 1999), 47–53.

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

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