Journal of Parallel and Distributed Computing **62**, 982–999 (2002)
doi:10.1006/jpdc.2001.1833

A Stabilizing Algorithm for Finding Biconnected Components

Mehmet Hakan Karaata

Department of Computer Engineering, Kuwait University, P.O. Box 5969, Safat 13060, Kuwait
E-mail: karaata@eng.kuniv.edu.kw

Received December 27, 1999; accepted January 11, 2002

In this paper, a self-stabilizing algorithm is presented for finding biconnected components of a connected undirected graph on a distributed or network model of computation. The algorithm is resilient to transient faults, therefore, it does not require initialization. The proposed algorithm is based on stabilizing BFS construction and bridge-finding algorithms. Upon termination of these algorithms, the proposed algorithm terminates after $O(d)$ rounds, where d is the diameter of the biconnected component with the largest diameter in the graph. The paper concludes with remarks on issues such as the adaptiveness of the algorithm.

Key Words: biconnected components; distributed systems; fault-tolerance; self-stabilization.

1. INTRODUCTION

Consider a connected undirected graph $G = (V, E)$, where V is the set of n nodes, and E is the set of edges. Without loss of generality, we assume V to be $\{1, 2, \ldots, n\}$. Two edges e and e' are said to be B-connected, denoted by eBe', if $e = e'$ or there exists a *simple cycle* containing both e and e'. A simple cycle is a path $(u, v, \ldots, u)$ such that no vertex except u in the path occurs twice. *B-connected* is an equivalence relation which partitions the edges of G into equivalent classes such that two distinct edges are in the same class iff they lie on a common simple cycle. Let E_i be such an edge class and V_i be the set of nodes (vertices) incident on the edges in E_i; subgraph $G_i = (V_i, E_i)$ is referred to as a *biconnected component* (BCC) of G. Informally, a connected graph is *biconnected* if it contains no cut-vertex, that is, a vertex whose removal disconnects the graph. Each maximal connected subgraph of G without a cut vertex is referred to as a biconnected component of G.

If the graph represents a communication network, then nodes in the same biconnected component can enjoy a greater degree of fault tolerance in their

0743-7315/02 $35.00

communication in the network. In addition, a biconnected component is less prone to the problem of network partitioning. Thus, from the fault tolerance point of view, identification of the biconnected components of a network is essential because biconnected components can tolerate a node failure in the network, while preserving at least one path between any two non-failed nodes in each biconnected component. Furthermore, many distributed algorithms for combinatorial problems on graphs assume that the graph is biconnected; therefore, the identification of biconnected component is essential [14].

A highly desirable property of fault-tolerant distributed systems is the property of self-stabilization. A self-stabilizing system guarantees that, regardless of the current state, the system reaches a legal state in a finite number of steps and the system state remains legal thereafter. Since the introduction of self-stabilization by Dijkstra [10], primarily in distributed systems, self-stabilizing algorithms for many fundamental problems in distributed systems have been proposed. For example, self-stabilizing mutual exclusion algorithms for a variety of network classes have been presented [6, 8, 10]. Self-stabilizing algorithms for electing a leader appear in [17, 22]. Self-stabilizing algorithms for a variety of graph theoretic problems are presented in [7, 13, 20, 19]. General techniques for constructing self-stabilizing algorithms are dealt with in [3, 4, 21]. Self-stabilizing algorithms are able to withstand *transient failures*. In addition, many self-stabilizing algorithms are capable of dealing with the dynamic addition and deletion of vertices or edges. Schneider provides a detailed survey on self-stabilization [23].

The problem of finding biconnected components has been studied extensively. A sequential algorithm for finding biconnected components is presented in Baase [5]. Static and dynamic distributed solutions are given in [2, 9, 15, 16, 24, 25]. However, there does not exist any self-stabilizing algorithm for finding biconnected components in the literature. In this paper, we present the first self-stabilizing distributed algorithm for finding biconnected components of a graph in the presence of *transient faults*. We view a fault that perturbs the state of the system but not the program as a transient fault. After a transient fault occurs and its effects are seen in the system, the algorithm eventually recovers from the fault. Since the algorithm is stabilizing, it does not require initialization. The proposed algorithm, referred to as Algorithm BCC, requires $O(d)$ rounds to identify all biconnected components in a graph upon termination of the underlying layers, where d is the diameter of the biconnected component with the largest diameter in the graph.

It is possible to obtain a stabilizing algorithm for finding biconnected components using the general technique of Dolev and Herman [11]. Their general mechanism can take any silent task and construct a stabilizing algorithm for the task. However, general techniques often require a global state collection followed by a distributed reset. Hence, they require a larger state space and time complexity than their handcrafted counterparts. Our algorithm identifies each biconnected component locally and independently in $O(d)$ rounds, where d is the diameter of the biconnected component with the largest diameter as opposed to the general mechanism of Dolev and Herman which requires a global state collection which takes $O(D)$ rounds, where D is the diameter of the graph. For this reason, our algorithm is more efficient with respect to time and state complexity than the one constructed by the general

mechanism of Dolev and Herman. The same applies to the other aforementioned general techniques for constructing stabilizing algorithms.

The paper is organized as follows. In Section 2, we present our model of computation. Section 3 contains some definitions and provides the basis of the algorithm. Section 4 presents the self-stabilizing algorithm for finding biconnected components. Section 5 includes the correctness proof of the proposed algorithm and a discussion on fairness issues related to the algorithm. We conclude the paper in Section 6 with remarks on related issues such as time complexity.

2. COMPUTATIONAL MODEL

Let $G = (V, E)$ be an arbitrary graph with node (vertex) set V and edge set E, where $|V| = n$. We assume that each node of G is a process with a unique id, also called node. Each node maintains a set of local variables whose values can be updated by the node after inspecting its own local variables and the local variables of its neighbors. The program of a node i can be expressed as

$$*[G[1] \rightarrow M[1] \quad G[2] \rightarrow M[2] \quad \Box \ldots \Box \quad G[k] \rightarrow M[k]],$$

where

- each *guard* $G[\ ]$ is a boolean function of the variables of node i and the variables of its neighboring nodes;
- each *move* $M[\ ]$ updates the variables of node i; and
- $*[S]$ corresponds to the repeated execution of the statement S until all guards are false.
- $\Box$ is called the non-determinism symbol. One of the guarded actions separated by them is selected non-deterministically in each iteration.

Note that each move denotes an atomic action. If a guard of node i evaluates to true, we say that the guard and node i are enabled. Furthermore, if two or more guards of the same node or of different nodes are enabled at the same time, then the *scheduler* arbitrarily selects a subset of the enabled guards and allows the execution of the corresponding moves to be completed. In the case of the *central scheduler*, the size of the subset is one and the execution of a move is completed before any guard is reevaluated; whereas in the case of the *distributed scheduler*, the size of the set is arbitrary. We assume the presence of a central scheduler.

We assume that the scheduler is weakly fair. That is, if a guard is enabled and remains enabled, it eventually executes the corresponding action (move). We make this assumption for the sake of simplifying the proofs. As we show later, Algorithm BCC is correct without any fairness assumption. Note that the chosen hypothesis, central unfair scheduler, is compatible with that of the BFS and the bridge-finding algorithms.

The *state* of a node is composed of the set of variables at the node. The *system state* is the cartesian product of the states of the nodes in the system. A *state* of the

system is an element of the state space. The system state before the system is started is referred to as the *initial state*.

3. BASIS OF THE ALGORITHM

3.1. Stabilizing BFS Construction and Bridge Finding

Prior to formally describing the algorithm, we first present the basis of the algorithm. The algorithm is composed of three layers. The first (lowest) layer consists of a self-stabilizing algorithm for constructing a breadth-first search (BFS) tree of a connected undirected graph $G = (V, E)$. The second layer is the stabilizing bridge-finding algorithm. The third (highest) layer is the proposed algorithm referred to as BCC algorithm. We assume that the BFS spanning tree of G, rooted at $r \in V$, is first obtained by applying one of the self-stabilizing BFS algorithms described in [1, 3, 12, 18]. The other layers of the algorithm assume that a spanning tree of the given graph $G = (V, E)$ is available. Therefore, in the rest of this paper, we assume the presence of the BFS spanning tree $T(r) = (V, E')$ rooted at node $r \in V$, and oriented from the root to the leaves, where $E' \subset E$ constructed by a self-stabilizing BFS algorithm as mentioned above.

We need the following definitions to describe the second layer of the algorithm. A node i in V is said to be the child of another node j, with respect to spanning tree $T(r)$, if j is the immediate predecessor of i in $T(r)$; alternatively, j is said to be the parent of i. Similarly, a node i in V is said to be a descendent of another node j, with respect to spanning tree $T(r)$, if j is a predecessor of i in $T(r)$; alternatively j is said to be an ancestor of i. For each node $i \in V$, $f(i)$ denotes the parent of node i in $T(r)$, and $f(r)$ denotes r indicating that root r is its own parent. For a pair of nodes i and j in a rooted tree $T(r) = (V, E')$, the collection of nodes is defined to be the set of *common ancestors* $\{x_1, x_2, \ldots, x_p\}$ if and only if each node x_k, $0 < k \leqslant p$, is an ancestor of both i and j. It is obvious that the root of the tree $r \in \{x_1, x_2, \ldots, x_p\}$. A common ancestor v of a pair of nodes i and j in a rooted tree $T(r)$ is defined to be the *lowest common ancestor*, denoted by $lca(\{i, j\})$, if and only if every common ancestor $u \in \{x_1, x_2, \ldots, x_p\} \backslash \{v\}$ of i and j is an ancestor of v. Let edge $\{i, j\} \in E \backslash E'$ be a non-tree edge of G and $v \in V$ be the lowest common ancestor of nodes i and j in $T(r) = (V, E')$. Also, let node disjoint paths (disjoint except for the terminals) B_i and B_j, with origins i, j and terminals v, v in V, respectively, be two paths of G such that every two consecutive nodes on each of these paths are joined by a tree edge. Paths B_i and B_j are referred to as *link paths* of non-tree edge $\{i, j\}$. Notice that for each non-tree edge, the link paths connect the end points of the non-tree edge to their lowest common ancestor.

The second layer is made up of a self-stabilizing algorithm called the *self-stabilizing bridge-finding algorithm* computing values referred to as s-values and given in [20]. s-values computed by this algorithm can be used to find bridges of G and are also used for finding biconnected components of G in this paper. In the rest of this paper, we assume that s-values computed by the bridge-finding algorithm satisfy the following property after the termination.

For every node $v \in V$:

$\{i, j\} \in s(v)$ iff $\{i, j\}$ is a non-tree edge and v is the lca of nodes i and j.
$\{i, j\} \in s(v)$ iff $\{i, j\}$ is a non-tree edge and v is not the lca of nodes i and j, and v is on the link path B_i or B_j.

The above concepts are illustrated with the help of an example in Fig. 1. In the figure, a BFS tree rooted at node 1 is shown where each tree edge is shown by a solid line and each non-tree edge is shown by a dashed line. The graph contains two non-tree edges, namely $\{3, 4\}$ and $\{6, 7\}$. Notice that the s-set of each ancestor of nodes 3 and 4 up to their lowest common ancestor contains $\{3, 4\}$. Since node 1 is the lowest common ancestor of nodes 3 and 4, its s-set contains $\{\{3, 4\}\}$. Similarly, since node 5 is the lowest common ancestor of nodes 6 and 7, its s-set contains $\{\{6, 7\}\}$. Also, notice that the only bridge in the graph is edge $\{4, 5\}$ and s-sets of nodes 4 and 5 do not contain any element in common indicating that edge $\{4, 5\}$ is a bridge. It is easy to see that every other pair of nodes $\{i, j\}$ such that $\{i, j\} \in E'$ contains a common element in their s-sets. (We say that two nodes contain a common element x when either both the nodes contain x or one contains x and the other contains $\{x\}$.)

We assume that the self-stabilizing BCC algorithm and the other two layers of the algorithm run concurrently; however, before the BFS tree is constructed, the moves made by the layers above may not guarantee any progress. Similarly, before the BFS tree is constructed and the s-values are computed, the moves made by the third layer may not guarantee any progress. Upon termination of the BFS algorithm, the second layer computes the s-values using the BFS tree. Analogously, after termination of the layers below, the third layer of the algorithm finds the biconnected components using the s-values and the BFS tree. For further discussions on the compositions of stabilizing algorithms, one may refer to [12]. We are not presenting the self-stabilizing BFS algorithm and the self-stabilizing bridge-finding algorithm, since they can be adapted from [1, 3, 12, 18] and [20], respectively.

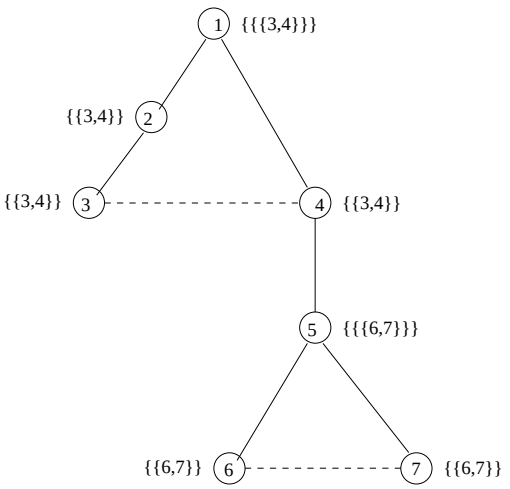

FIG. 1. The state of a system after termination.

3.2. The Approach

In order to facilitate the description of the third layer of the algorithm, we need to make the following definitions.

Let i, j be two nodes connected by a non-tree edge. Also, let v be the lowest common ancestor of nodes i and j. The paths over tree edges from i to v, and from j to v and the non-tree edge $\{i, j\}$ make up a *fundamental cycle* (FC) of non-tree edge $\{i, j\}$. The fundamental cycle formed by non-tree edge $\{i, j\}$ is denoted by $FC(\{i, j\})$. $FC(\{i, j\})$ also denotes the set of nodes in the fundamental cycle formed by non-tree edge $\{i, j\}$. Node v, the lowest common ancestor of nodes i and j, is referred to as the *lca* of $FC(\{i, j\})$ and denoted by $lca(\{i, j\})$. Also, $lca(x)$ denotes the lowest common ancestor of endpoints of the non-tree edge that form the fundamental cycle x. It is easy to see that there are as many fundamental cycles as the number of non-tree edges in G. Two FCs u, w are said to be *neighboring* iff they share a common edge. Two FCs u, w are said to be *neighboring with respect to node* $i \in FC(u) \cup FC(w)$ iff they share a common edge and i is an ancestor of $lca(u)$ or $lca(w)$. Furthermore, two FCs u_1, u_k are said to be *transitively connected* iff there exists a sequence of FCs $u_1, u_2, \ldots, u_k$, where $1 < k < n$, such that each pair of FCs u_l, u_{l+1}, for $0 < l < k$, in the sequence are neighboring fundamental cycles. Similarly, two FCs u_1, u_k are said to be *transitively connected with respect to node* i iff there exists a sequence of FCs $u_1, u_2, \ldots, u_k$, where $1 < k < k$, such that each pair of FCs u_l, u_{l+1}, for $0 < l < n$, in the sequence are neighboring with respect to node i.

Using the aforementioned property of s-values, the nodes contained in the fundamental cycle formed by non-tree edge $\{i, j\}$ can easily be identified because each node in the fundamental cycle contains either $\{i, j\}$ or $\{\{i, j\}\}$ in its s-value.

The self-stabilizing BCC algorithm is based on the following two important observations. These are presented in the form of the following lemmas, whose proofs are straightforward and hence omitted.

LEMMA 1. *Two FCs u, w belong to the same biconnected component iff u and w are transitively connected.*

LEMMA 2. *If two fundamental cycles u and w are neighboring, then the lca of u is a descendent of that of w, the lca of w is a descendent of that of u, or their lca's are the same node in T(r).*

Lemma 1 can be used to find whether or not two nodes x and y are in the same biconnected component. If x and y are contained in fundamental cycles u and w, respectively, and u and w are transitively connected, then nodes x and y are in the same biconnected component. Otherwise, they are not.

Lemma 2 describes the three possible orientations of two neighboring fundamental cycles. The proposed algorithm considers all these orientations in discovering that two fundamental cycles are neighboring. Consequently, this neighborhood knowledge can be used to discover the transitively connected fundamental cycles.

We now describe our approach implemented by algorithm BCC to identify the biconnected components of G. We associate a unique id with each edge in G. In addition, each nodal process maintains a variable b called b-set of node i and

denoted by $b(i)$ containing a set of tuples. Informally, starting from the nodes at the largest depth of each biconnected component B, the b-set of each node i in B collects the set of descendants of i contained in the biconnected component containing i in a bottom-up fashion. Eventually, the b-set of the *ancestor* of each biconnected component B contains the set of its descendents in the biconnected component. If i is a node in biconnected component B such that each node in $B\backslash\{i\}$ is a descendent of i, then node i is referred to as the ancestor of biconnected component B. The fundamental data structure b-set of the algorithm is not a simple set, and therefore requires further explanation. We now describe b-sets in more detail. Each tuple in $b(i)$ is of the form $\{x, y\}$, where $x = \{p, q\}$ denotes a non-tree edge incident on a descendent of i joining nodes p and q, and y denotes a set of descendants of i. Upon termination of the algorithm, the following holds. For each node $i \in V$, $\{\{p, q\}, y\} \in b(i)$ iff ($\{p, q\} \in s(i)$ or $\{\{p, q\}\} \in s(i)$) holds, and y is the set of descendants of i except p and q in the fundamental cycle $FC(\{p, q\})$ and in the FCs whose lcas are a descendent of i and transitively connected to $FC(\{p, q\})$ with respect to i.

After the b-sets are computed in a bottom up fashion and the above condition holds, the biconnected components can be identified as follows. If $FC_1, FC_2, \ldots, FC_k$ is a maximal sequence of transitively connected fundamental cycles such that fundamental cycles FC_j and FC_{j+1} for $0 < j < k$ are neighboring, the set of nodes in $B = FC_1 \cup FC_2 \cup \cdots \cup FC_k$ is a biconnected component of G by Lemma 1. Observe that if the above condition holds after termination, then we have the following. For each node i in G that is an ancestor of a biconnected component B, such that i is the lca of $FC(x_1), FC(x_2), \ldots, FC(x_l)$, where each $FC(x_j)$, for $1 \leqslant j \leqslant l$, is one of the maximal sequence of transitively connected fundamental cycles $FC_1, FC_2, \ldots, FC_k$, then the following holds:

$$\{x_1, y_1\}, \{x_2, y_2\}, \ldots, \{x_l, y_l\} \in b(i) \text{ iff } B = \{i\} \cup x_1 \cup y_1 \cup x_2 \cup y_2 \cup \cdots \cup x_l \cup y_l.$$

Using the above, upon termination, the ancestor of each biconnected component identifies the nodes in the biconnected component. Once a biconnected component is identified by its ancestor, using a separate mechanism, i may broadcast its knowledge to each node in the biconnected component.

We use the s-values computed by the second layer of the algorithm to determine the number of tuples and the first element of each tuple in each b-set by ensuring that for $i \in V$, $\{x, \_\} \in b(i)$ iff $x \in s(i)$ or $\{x\} \in s(i)$ holds, where $\{x, \_\}$ is a tuple, whose first element is x and the second element is arbitrary. After the s-values are computed by the second layer of the algorithm, the first element of each tuple can readily be computed.

The main task of the third layer of the algorithm is to compute the second element of each tuple. After for $i \in V$, $\{x, \_\} \in b(i)$ iff $x \in s(i)$ or $\{x\} \in s(i)$ holds, we know that each node i in fundamental cycle $FC(\{p, q\})$ has tuple $\{\{p, q\}, \_\}$ in $b(i)$. Recall that $\{\{p, q\}, \_\}$ denotes a tuple whose first element is $\{p, q\}$ and the second element is arbitrary. Using this information, we can readily identify each fundamental cycle by collecting the node id's of all the nodes in the FC in a bottom up fashion. In this process, each node $i \in V$ contained in the fundamental cycle $FC(x)$ updates a given element $\{x, y\}$ in $b(i)$ as follows: for each child $j \in V$ of i such that $\{x, z\} \in b(j)$, it adds

j to y only if $j \notin x$ and $j \notin y$, and it adds $k \in z$ to y only if $k \notin x$ and $k \notin y$. The condition $j \notin x$ ensures that the endpoints of non-tree edges are not added and the condition $j \notin y$ prevents addition of nodes, which are already contained in y. Eventually, the lca of each FC knows all the nodes in the FC. That is, if node i is the lca of $FC(x)$, eventually $x \cup y \supseteq FC(x)$ holds, where $\{x, y\} \in b(i)$. Observe that since an edge is defined as a set with two elements, we are able to use the set of operations on edges such as $i \in x$, where x is an edge in G.

Now, we are to describe the way in which a node knows about the set of its descendants contained in fundamental cycles transitively connected to the fundamental cycles it is contained in. If two FCs u and w formed by non-tree edges $\{i, j\}$ and $\{p, q\}$, respectively, are neighboring and have a common lca, then the lca knows all the nodes in u and w in the aforementioned manner. If FCs u and w are such that the lca of w is a descendent of that of u, the lca of u eventually knows all the nodes in u and w as follows. We know that $lca(w)$ eventually knows all the nodes in w. We also know that the parent k of $lca(w)$ eventually knows all its descendants D in u. By the definition of neighboring fundamental cycles, we know that D and nodes in w have a common element. Node k eventually discovers that D and w have a common node by examining its state and the state of its child $lca(w)$ and includes nodes in w into y, where $\{\{i, j\}, y\} \in b(k)$. At this point, node k knows all its descendants in u and all the nodes in w. Then, the parent $p.k$ of k includes nodes in w into y', where $\{\{i, j\}, y'\} \in b(p.k)$, and so on. Starting from node k, node id's of nodes in w propagate up to l, where l is $lca(u)$. That is, on the path $lca(w), v_1, v_2, \ldots, lca(u)$ in $T(r)$, each node eventually knows the nodes in w and its descendants in u. Then, if u has a neighboring fundamental cycle z whose lca is higher in $T(r)$ than those of w and u, $lca(z)$ eventually knows the nodes in w, u and z. In this manner, the node id's of transitively connected component are gathered and propagated upwards to the ancestor of each biconnected component. When the algorithm terminates, for each biconnected component B, the ancestor of B knows all the nodes in B. It is easy to see that this information can be disseminated to all the nodes in B. However, for the sake of brevity, we do not present the implementation of this process of dissemination.

3.3. Illustrative Example

The above concepts are illustrated with the help of an example in Fig. 2. In the figure, a graph with a BFS tree rooted at node 11 is shown where each tree edge is shown by solid lines and each non-tree edge is shown by dashed lines. The graph contains four non-tree edges, namely $\{1, 2\}$, $\{4, 5\}$, $\{6, 7\}$ and $\{7, 8\}$. Observe that fundamental cycles $FC(\{1, 2\}) = \{1, 2, 10\}$, $FC(\{4, 5\}) = \{3, 4, 5, 6, 10\}$, $FC(\{6, 7\}) = \{6, 7, 9, 10\}$ and $FC(\{7, 8\}) = \{7, 8, 9, 10, 11\}$ exist in the graph. Notice that $FC(\{1, 2\})$ does not have a neighboring FC, whereas, $FC(\{4, 5\})$ and $FC(\{6, 7\})$, and $FC(\{6, 7\})$ and $FC(\{7, 8\})$ are neighboring. Also notice that FCs $FC(\{4, 5\})$, $FC(\{6, 7\})$ and $FC(\{7, 8\})$ are pairwise transitively connected. It is easy to see that each tuple $\{x, y\} \in b(i)$ indicates the set of descendants of i in $FC(x)$ and in FCs transitively connected to $FC(x)$ with respect to i. For instance, $b(10)$ includes $\{\{1, 2\}, \{\}\}$, where $\{\}$ denotes the empty set, indicating that nodes 10 and its descendants 1, 2, are in $FC(\{1, 2\})$. Also, $b(10)$ includes $\{\{4, 5\}, \{3, 6\}\}$ indicating that node 10 and its descendants 3, 4, 5, 6 are

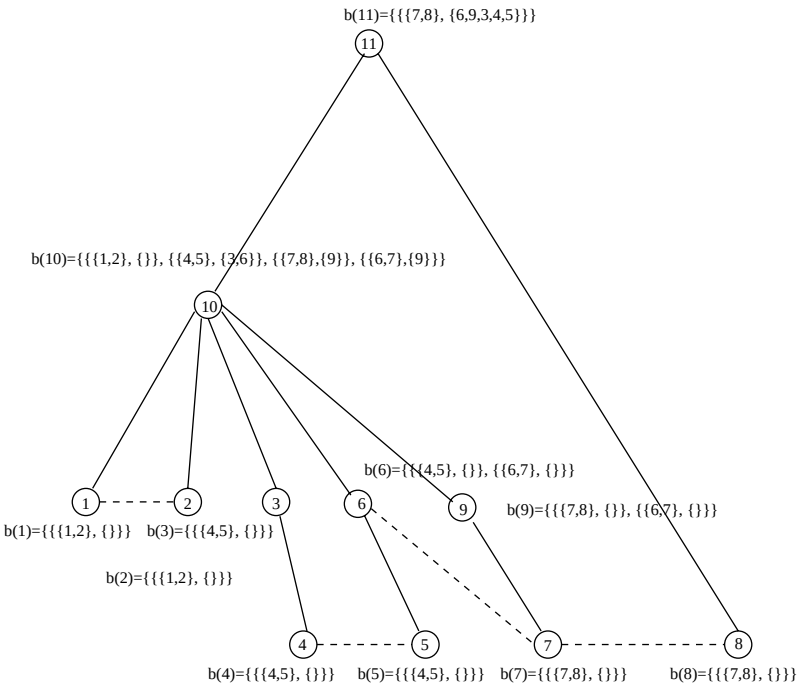

FIG. 2. The state of a system after termination.

in $FC(\{4,5\})$ and in FCs transitively connected to $FC(\{4,5\})$ with respect to node 10. Similarly, $b(11)$ includes $\{\{7,8\},\{3,4,5,6,9\}\}$ indicating that node 11 and its descendants $3,4,5,6,9$ are in $FC(\{7,8\})$ and in FCs $FC(\{4,5\})$ and $FC(\{6,7\})$ that are transitively connected to $FC(\{7,8\})$ with respect to node 11. Also, notice that among the FCs $FC(\{4,5\})$, $FC(\{6,7\})$ and $FC(\{7,8\})$ that are transitively connected, $FC(\{7,8\})$ has the highest lca node 11 in $T(11)$. Since $b(11)$ contains $\{\{7,8\},\{3,4,5,6,9\}\}$, nodes $3,4,5,6,7,8,9$ are in a biconnected component. However, since $b(11)$ does not include $1,2$ in any tuple, nodes $1,2$ and node 11 are not in the same biconnected component. It is easy to see that for every other node i, $\{x,y\} \in b(i)$ iff $(\{p,q\} \in s(i)$ or $\{\{p,q\}\} \in s(i))$ holds, and set y denotes the set of descendants of i in the fundamental cycle $FC(x)$ and in the FCs whose lcas are descendent of i and are transitively connected to $FC(x)$ with respect to i.

4. STABILIZING ALGORITHM FOR FINDING BICONNECTED COMPONENTS

In this section, we present the self-stabilizing algorithm for finding biconnected components implementing the strategy described above. We first introduce a function to facilitate the description of the algorithm:

$b(i)$ denotes a set of tuples of the form $\{x,y\}$, where $x = \{p,q\}$ denotes a non-tree edge incident on a descendent of i joining nodes p and q. $b(i)$ is also referred to as the b-set of node i. For node $i \in V$, $\{\{p,q\},y\} \in b(i)$ iff $(\{p,q\} \in s(i)$ or

$\{\{p,q\}\} \in s(i)$) holds, and y is the set of descendants of i except p and q in the fundamental cycle $FC(\{p,q\})$ and in the FCs whose lca's are descendent of i and are transitively connected to $FC(\{p,q\})$ with respect to i.

We also need the following notation:

$a^0.x$ at node i denotes the set of descendants of i in the fundamental cycle $FC(x)$ and in the fundamental cycles neighboring $FC(x)$ with respect to i.

$a^k.x$ at node i denotes the set of descendants of i in the fundamental cycles $FC_0, FC_1, \ldots, FC_k$ such that $FC_0 = FC(x)$, for $0 \leqslant l \leqslant k$, each FC_l is transitively connected to $FC(x)$ with respect to i, and for $0 \leqslant l < k$, FC_l and FC_{l+1} are neighboring with respect to i.

$a.x$ at node i denotes the set of descendants of i except p and q in the fundamental cycle $FC(x)$ and in the FCs transitively connected to $FC(x)$ with respect to i.

Notice that each one of the decisions mentioned above is based on the local view of each node made up of its state and neighboring states. Therefore, due to having initial arbitrary values, at some point in time the local view of each node may not reflect the reality. The goal of the proposed algorithm is to ensure that eventually each node's view reflects the reality.

The self-stabilizing algorithm for finding biconnected components is given in Fig. 3. The formal definitions of $a^0.x$, $a^k.x$ and $a.x$ are given along with the algorithm based on the state of node i and its neighbor's states.

Throughout the paper, we refer to the first guard of the algorithm as $G1$, the second guard as $G2$. Upon discovering that for node i, $\{x\} \in s(i) \vee x \in s(i)$ holds, node i includes $\{x, a.x\}$ in $b(i)$. Note that $a.x$ for node i includes the set of descendants of node i in $FC(x)$ and in the fundamental cycles transitively connected to $FC(x)$ with respect to i except nodes in x according to node i's current view of the system. Each node id $i \in V$ contained in $FC(x)$, propagates up to the lca v of the FC with the highest lca among the FCs transitively connected to $FC(x)$ in $T(r)$. This propagation is carried out by including i in the b-sets of its ancestors up to v as follows. If j is in $FC(x)$, then the parent j of i includes i in tuple $\{x, y\}$ in $b(j)$. When we say that node id i is included in tuple $\{x, y\}$, we mean that i is included in y if it is not already in x or y. Then, the parent $p.j$ of node j includes i in tuple $\{x, y'\}$ in $b(p.j)$, and so on. Eventually, i is included in tuple $\{x, y''\}$ in the b-set of the lca of $FC(x)$. Consequently, i is included in the b-set of the lca of the FC whose lca is the lowest ancestor of i among the lca's of the FC's transitively connected $FC(x)$, then, the second lowest, and so on. Eventually, i is included in the b-set of the ancestor of the biconnected component containing i. Notice that this propagation is implemented by $G1$ of Algorithm BCC. G2 ensures that the values contained in the b-sets due to arbitrary initialization and the moves caused by arbitrary initialization are removed.

Thus far, we have presented an algorithm that computes b-sets for each node. Note that since the information about the biconnected components in the system is implicitly maintained in b-sets of nodes in a distributed manner, therefore, we do not explicitly identify the biconnected components. Based on the b-sets, when the

Node i

VARIABLE $s(i)$: a set of elements of the form x and {x}, where x is an edge in G

VARIABLE $b(i)$: a set of elements of the form $\{x, y\}$, where x is an edge in G and y is a set of nodes in G

PARAMETER x : an edge in G

ACTIONS

$$
\begin{array}{lll}
*\Big[\{x\}, x \in s(i) \wedge \{x, a.x\} \notin b(i) & \longrightarrow & b(i) = b(i) \setminus \{x, \_\} \cup \{\{x, a.x\}\} \\
\square\ (\{x\}, x \in s(i) \wedge \{x, y\} \in b(i) \wedge y \neq a.x) \vee & & \\
(x \notin s(i) \wedge \{x\} \notin s(i) \wedge \{x, y\} \in b(i)) & \longrightarrow & b(i) = b(i) \setminus \{x, \_\}\Big]
\end{array}
$$

where

$a^0.x$	: $x \cup \bigcup_{j \in c.i}(\{j\} \cup z \mid x \in s(i) \wedge \{x, z\} \in b(j))$
$a^k.x$	: $\{z \cup l \cup a^{k-1}.x \mid \exists_{j \in c.i}(\{z, l\} \in b(j) \wedge (\{z \cup l\} \cap a^{k-1}.x \neq \emptyset))\}$
$a.x$	: $a^k.x - x$ such that $a^k.x = a^{k-1}.x$
$c.i$	: denotes the set of immediate descendants, children, of node i.
$\{x, \_\}$	: denotes a tuple whose first element is x but the second element is arbitrary.
$\emptyset$	: denotes the empty set.

FIG. 3. The self-stabilizing algorithm for finding biconnected components (Algorithm BCC).

algorithm terminates, for each biconnected component B, the ancestor B knows all the nodes in B. Consequently, each node in B may obtain this information and agree on the nodes in B.

5. CORRECTNESS

In this section, we show that Algorithm BCC is correct and self-stabilizing.

5.1. Safety and Liveness

Let *PRE* be a predicate defined over *SYS*, the set of global states of the system. An algorithm *ALG* running on *SYS* is said to be *self-stabilizing* with respect to *PRE* if it satisfies:

Safety: If a global state q satisfies *PRE*, then any global state that is reachable from q using algorithm *ALG*, also satisfies *PRE*.

Liveness: Starting from an arbitrary global state, the distributed system *SYS* is guaranteed to reach a global state satisfying *PRE* in a finite number of steps of *ALG*.

Global states satisfying *PRE* are said to be *stable*. Similarly, a global state that does not satisfy *PRE* is referred to as an *unstable state*. To show that an algorithm is self-stabilizing with respect to *PRE*, we need to show the satisfiability of both safety and liveness conditions. In addition, to show that an algorithm solves a certain problem, we need to either prove *partial correctness* or show that through transitions made by the algorithm among stable states the problem is solved.

We now show that Algorithm BCC is self-stabilizing by establishing the liveness and the safety properties.

Let P be a predicate defined as follows. For node $i \in V$, $\{\{p, q\}, y\} \in b(i)$ iff ($\{p, q\} \in s(i)$ or $\{\{p, q\}\} \in s(i)$) holds, and y is the set of descendants of i except p and q in the fundamental cycle $FC(\{p, q\})$ and in the FCs whose lcas are descendent of i and transitively connected to $FC(\{p, q\})$ with respect to i.

Now, we present the worst case time complexity or the upper bound of the BCC algorithm.

We assume that each node executes all its enabled actions in a step referred to as a *move*. We first classify the moves of the algorithm into two categories called *initial moves* and *non-initial moves*. The initial moves are those that are caused by arbitrary initialization and the non-initial moves are those that are caused by other moves in the system. We categorize each move as an initial move or as a non-initial move as follows.

Move M_x by node i is a non-initial move if there exits a move M_y by a child j of i such that move M_y happens before move M_x and move M_x is not *enabled* prior to move M_y. Otherwise, a move is referred to as an initial move. We say that a move is enabled if the conditions are satisfied for the move to take place. Let M_y be the last such move. Move M_y is referred to as the cause of move M_x.

An execution in a distributed system can be described as a sequence of moves $M_1, M_2 \ldots$, where M_j is a move made by a node in the system. Consider move M_x by an arbitrary node i. We identify a unique node i and a unique move M_y, where $l < k$, by node j which is the "cause" of move M_x defined as follows.

Define *cause*( ) for initial moves

- $cause(M_x) = M_x$ if M_x is an initial move.

Define *cause*( ) for non-initial moves

- $cause(M_x) = M_y$ if M_y is a move such that move M_y happens before move M_x and move M_x is not *enabled* prior to move M_y.

We now state several useful properties related to the function *cause*(). The first property is that distinct moves by a node have distinct causes.

PROPOSITION 1. *If M_p and M_q are distinct moves by node i then*

$$cause(M_p) \neq cause(M_q).$$

PROPOSITION 2. *A move by a node can only cause a move by its parent.*

The next property that we establish is that the cause relationship is "acyclic". The following proposition follows from the definition of cause.

PROPOSITION 3. *Let M_x be a move by node i. If M_x is not an initial move by node i, then the move $cause(M_x) = cause(cause(M_x))$ is not made by node i.*

We now show the upper bound on the number of initial moves.

LEMMA 3. *The total number of initial moves in the system is n.*

Proof. By the definition of a move, observe that after a node takes its initial move, it can no longer make an initial move. Hence, the proof follows. ∎

Now, based on the definition of *cause*( ) we define the notion of the source of a move. We then show that distinct moves have distinct sources and use that to prove an upper bound on the total number of moves made by all the nodes in the system.

For each move M_k define

$$source(M_k) = \begin{cases} M_k & \text{if } M_k \text{ is an initial move,} \\ source(cause(M_k)) & \text{if } M_k \text{ is a non-initial move.} \end{cases}$$

Intuitively, $source(M_k)$ can be thought of as an initial move that causes M_k through a chain of moves.

The following proposition states a useful property of sources of moves. The proof of this proposition immediately follows from Propositions 1 and 3; hence it is omitted.

PROPOSITION 4. *The source of each move of Algorithm SCC is a distinct initial move.*

Each initial move can trigger a bounded number of moves. The following lemma provides an upper bound on the number of moves caused by a single source. Since a move by a node can only cause a move by its parent as stated by Proposition 2 and the fact that G contains n nodes, we have the following proposition.

PROPOSITION 5. *Each distinct initial move can be the source of at most* $n - 1$ *moves.*

The following lemma establishes the liveness of the algorithm.

LEMMA 4 (Liveness). *All guards of algorithm BCC are disabled after* $O(n^2)$ *moves.*

Proof. It is easy to see that the proof follows from Lemma 3, and Propositions 4, 5. ∎

Having established the liveness of the algorithm, we now show the safety of the algorithm. For this purpose, we present the following lemma whose proof follows from the definition of *a.x* and the description of the algorithm.

PROPOSITION 6. *Once predicate P holds, all guards of algorithm BCC are disabled.*

The safety property follows from Proposition 6 since all guards are disabled once P holds. Therefore, the safety property is trivially satisfied. Hence, Algorithm BCC is self-stabilizing as stated by the following lemma whose proof follows from Lemma 4 and the above discussion.

LEMMA 5. *Algorithm BCC is stabilizing.*

5.2. Partial Correctness

The following lemma establishes the partial correctness of our algorithm.

LEMMA 6. *If all guards of algorithm BCC are disabled, after they are all disabled, predicate P is satisfied.*

Proof. By induction on the height h of i from p or q, where $x = \{p, q\}$ whichever is larger.

Induction hypothesis: For each node $i \in V$ and for each $\{p, q\} \in s(i)$ and $\{\{p, q\}\} \in s(i)$, where, such that i is at height h from p or q whichever is larger, predicate P holds.

Basis: $h = 0$.

Notice that, in this case, $i = p$ or $i = q$ holds. Clearly, for each $\{p, q\} \in s(i)$, $a.x = \emptyset$, where $x = \{p, q\}$ and $\{\{x, y\}\}, \emptyset\} \in b(i)$ for node i hold. Hence, P is satisfied.

Induction step: Assume the induction hypothesis. To show that for each node $i \in V$ and for each ($\{p, q\} \in s(i)$ or $\{\{p, q\}\} \in s(i)$) such that i is at height $h + 1$ from p or q whichever is larger, predicate P is satisfied. Since no guard is enabled at i, we know that $b(i) = \{x, a.x\}$ holds. Let function $L(x, i)$ denote the set of descendants of i, where $x = \{p, q\}$, contained in $FC(x)$ and in all the fundamental cycles transitively connected to $FC(x)$ with respect to i. Let $i \in V$ be a node at height $h + 1$ from p or q whichever is larger such that ($\{p, q\} \in s(i)$ or $\{\{p, q\}\} \in s(i)$) holds, where $\{p, q\}$ is an arbitrary non-tree edge. Since $G1$ and $G2$ are disabled and ($\{p, q\} \in s(i)$ and $\{\{p, q\}\} \in s(i)$) hold, $\{\{p, q\}, y\} \in b(i)$. We need to show that $y = L(\{p, q\}, i)$. Also, let j be an arbitrary node in $L(x, i)$. Now, we show that x or $a.x$ at node i contains j. We know that one of the following cases must hold by Lemma 2.

Case 1: j is in $FC(\{p, q\})$.

Then, we know that the following holds by the induction hypothesis:

$$\exists_{k \in c.i} \exists_{\{p,q\} \in E \setminus E'} (\{p, q\} \in s(k) \wedge (\{p, q\} \in s(i) \vee \{\{p, q\}\} \in s(i)) \wedge \{\{p, q\}, y\} \in b(k) \wedge j \in y).$$

Clearly, by the definition of $a.x$, j is in $a.x \cup x$.

Case 2: j is contained in an FC that is transitively connected to $FC(\{p, q\})$ with respect to i and j is a descendent of $k \in c.i$. Without loss of generality, we assume that j is contained in an FC transitively connected to $FC(\{p, q\})$ with respect to i. Let $k \in c.i$ be such that k is on path from i to j over the tree edges. By the induction hypothesis, we have $\exists_{\{w,z\} \in b(k)} (j \in z)$. For nodes i and j one of the following must hold.

Subcase 1: $w = x$.

Notice that by the definition of $a.x$ and the fact that $G1$ is disabled, j is in $a.x$.

Subcase 2: $w \neq x$.

In this case, since $\{y, \_\} \notin b(i)$, we know that k is the lca of $FC(y)$ by Lemma 2. Since $FC(x)$ and $FC(w)$ are transitively connected with respect to i and by the induction hypothesis, there exists a sequence of tuples $c_1, c_2, \ldots, c_t$, where $0 < t < |E|$ such that $c_l = \{x_l, y_l\} \in b(m)$, where m is a child of i, $(x_l \cup y_l) \cap (x_{l+1} \cup y_{l+1}) \neq \emptyset$ for $0 < l < t$, $j \in (x_1 \cup y_1)$ and $\{x_t, y_t\} \in b(i)$ hold. Then, by the definition of $a.x$ and the fact that $G1$ is disabled, observe that j is in $a.x \cup x$.

To show if $\{x, y\} \in b(i)$, for an arbitrary node $j \in x \cup y$, $j \in L(x, i)$ holds and $x \in s \times (i) \vee \{x\} \in s(i)$ holds. Since $\{x, y\} \in b(i)$ and guards $G1$ and $G2$ are disabled, we know that $x \in s(i) \vee \{x\} \in s(i)$ holds. Now, we are to show $j \in L(x, i)$. We need to show that for each node $j \in x \cup y$, where $\{x, y\} \in b(i)$ such that i is height $h + 1$ p or q whichever is larger, $j \in L(x, i)$ holds. Since G2 is false and by the definition of $a.x$, we know that one of following cases holds. Now, we are to show that in each one of these cases $j \in L(x, i)$ holds.

Case 1: $j \in c.i \wedge \{x, \_\} \in b(j)$.

Since $\{x, z\} \in b(i), \{x, z\} \in b(j)$, we know that $j \in FC(p, q)$, where $x = \{p, q\}$. Therefore, by definition $j \in L(x, i)$ holds.

Case 2: $\exists_{k \in c.i} \exists_{\{x, z\} \in b(k)} (j \in z)$.

By the induction hypothesis and since $\exists_{k \in c.i} \exists_{\{x,z\} \in b(k)} (j \in z)$ holds, we know that j is contained in an FC transitively connected to $FC(\{p, q\})$ with respect to i, where $x = \{p, q\}$. Then, clearly $j \in L(x, i)$.

Case 3: There exists a sequence of tuples $c_1, c_2, \ldots, c_t$, where $0 < t < |E|$ such that $c_l = \{x_l, y_l\} \in b(m)$, where m is a child of i, $(x_l \cup y_l) \cap (x_{l+1} \cup y_{l+1}) \neq \emptyset$ for $0 < l < t$, $j \in (x_1 \cup y_1)$ and $\{x_t, y_t\} \in b(i)$ hold.

If $\{x_a, y_a\} \in b(c_a)$ and $\{x_b, y_b\} \in b(c_b)$, where $c_a, c_b \in c.i$ and $\{x_a \cup y_a\} \cup \{x_b \cup y_b\} \neq \emptyset$, then the fundamental cycles $FC(x_a)$ and $FC(x_b)$ are transitively connected by the induction hypothesis. Inductively, it can be readily shown that FCs $FC(x_1), FC(x_2), \ldots, FC(x_t)$ are all transitively connected with respect to i. Thus, $j \in L(x, i)$. Hence, the proof follows. ∎

5.3. Round Complexity and Correctness

We now provide the round complexity of our algorithm. A *round* refers to a minimum execution sequence in which each enabled action is taken at least once. In a round, an action that is disabled either before or after the round starts and remains disabled until the end of the round need not be executed. Let function $L(x, i)$ denote the set of descendants of i except p and q, where $x = \{p, q\}$, contained in $FC(x)$ and in the all the fundamental cycles transitively connected to $FC(x)$ with respect to i. The following lemma shows the round complexity of the BCC algorithm under the assumption that the underlying algorithms (layers) have already terminated.

LEMMA 7. *Algorithm BCC terminates after d rounds, where d is the diameter of the biconnected component with the largest diameter in G.*

Proof. It can readily be observed that after the first round, for each node $j \in V$, j has a non-tree edge x incident on it, iff it contains $\{x, \emptyset\}$ in $b(j)$ (see $G1$ and $G2$ of the algorithm).

In each one of the following rounds of the algorithm, for each node $j \in V$, node id j is guaranteed to propagate further up one level (from a child to its parent) in $T(r)$ until the propagation reaches the lca of the FC with the highest lca in $T(r)$ among the FCs that are transitively connected to the FC containing j. Observe that after k rounds of the algorithm, for every node $i \in V$ such that $d(i, j) \leqslant k$, if $j \in L(x, i)$, $\{x, y\} \in b(i) \wedge j \in y$ holds. Inductively, it can be shown that since the distance of the farthest node from i is d in the biconnected component containing i, after d rounds, the algorithm terminates. ∎

From Lemmas 5, 6 and 7, we have the following theorem.

THEOREM 1. *Algorithm BCC is self-stabilizing and it finds all biconnected components of G after $O(d)$ rounds upon termination of the underlying layers.*

We presented Algorithm BCC under the weak fairness assumption. The assumption of having an unfair scheduler (having no fairness assumption) means that an enabled action is guaranteed to be executed only when it remains as the only enabled action in the system. Otherwise, i.e., if there are other enabled actions in the system, the action execution may be deferred indefinitely. The following theorem shows that Algorithm BCC is correct without any fairness assumption.

LEMMA 8. *Algorithm BCC is correct without any fairness assumption.*

Proof. Suppose that there exits a guard which remains enabled but the corresponding move is never made. Then, the execution must continue with other enabled guards. Notice that each move by an arbitrary node i, disables its guard and may only enable a guard of node $f(i)$, parent of node i. As a consequence, since the graph does not contain any cycle, the number of moves in the system is finite. Therefore, the enabled guard eventually remains as the only enabled guard in the system and the corresponding move is made. Hence, the proof follows. ∎

6. CONCLUSIONS

It is desirable to devise a stabilizing algorithm for finding biconnected components that can deal with dynamic topology changes. In fact, the proposed algorithm can deal with dynamic addition/deletion of edges/vertices provided that the BFS tree construction and the stabilizing bridge-finding algorithms are dynamic, and the graph remains connected.

On a distributed or network model of computation, we have presented a self-stabilizing algorithm that identifies biconnected components in a connected undirected graph. We assumed that algorithm BFS of [18] is used to compute initially the BFS spanning tree $T(r)$ of G. We also assume that values called s-values satisfying the aforementioned property are computed by the self-stabilizing bridge-finding algorithm. The proposed algorithm can guarantee progress only after these two algorithms have terminated. Algorithm BCC is self-stabilizing, therefore, starting from an arbitrary initial state or upon a transient fault, it is possible to find the biconnected components of G after $O(d)$ rounds upon termination of the underlying layers. We showed that the proposed algorithm is correct and it works with an unfair scheduler.

ACKNOWLEDGMENTS

The author thanks his wife Özlem Karaata for her help in the preparation of the manuscript. We would also like to thank the anonymous referees and Bülent Müderrisoğlu for their suggestions and constructive comments on an earlier version of the paper. Their suggestions have greatly enhanced the readability of the paper.

REFERENCES

1. S. Aggarwal and S. Kutten, Time optimal self-stabilizing spanning tree algorithm, pp. 400–410, 1993.
2. M. Ahuja and Y. Zhu, An efficient distributed algorithm for finding articulation points, bridges and biconnected components in asynchronous networks, *in* "9th Conference on Foundations of Software Technology and Theoretical Computer Science, Bangalore, India," Lecture Notes in Computer Science, Vol. 405, pp. 99–108, Springer-Verlag, Berlin, 1989.
3. A. Arora and M. G. Gouda, Distributed reset, *IEEE Trans. Comput.* **43** (1994), 1026–1038.
4. B. Awerbuch and G. Varghese, Distributed program checking: a paradigm for building self-stabilizing distributed protocols, *in* "FOCS91 Proceedings of the 31st Annual IEEE Symposium on Foundations of Computer Science," pp. 258–267, 1991.
5. S. Baase, "Introduction to Design and Analysis of Algorithms," Addison–Wesley, Reading, MA, 1978.
6. G. M. Brown, M. G. Gouda, and C. L. Wu, Token systems that self-stabilize, *IEEE Trans. Comput.* **38** (1989), 844–852.
7. S. C. Bruell, S. Ghosh, M. H. Karaata, and S. V. Pemmaraju, Self-stabilizing algorithms for finding centers and medians of trees, *SIAM J. Comput.* **29** (1999), 600–614.
8. J. E. Burns and J. Pachil, Uniform self-stabilizing rings, *ACM Trans. Program. Languages Systems* **11** (1989), 330–344.
9. E. J. Chang, Echo algorithms: Depth parallel operations on general graphs, *IEEE Trans. Software* **8** (1982), 391–401.
10. E. Dijkstra, Self stabilizing systems in spite, of distributed control, *Comm. Assoc. Comput. Mach.* **17** (1974), 643–644.
11. S. Dolev and T. Herman, Superstabilizing protocols for dynamic distributed systems, *Chicago Journal of Theoretical Computer Science* **3** (1997).
12. S. Dolev, A. Israeli and S. Moran, Self-stabilization of dynamic systems assuming only read/write atomicity, *Distrib. Comput.* **7** (1993), 3–16.

13. S. Ghosh and M. H. Karaata, A self-stabilizing algorithm for coloring planar graphs, *Distrib. Comput.* **7** (1993), 55–59.
14. D. Hochbaum, Why should biconnected components be identified first, *Discrete Appl. Math. Combinat. Oper. Res. Comput. Sci.* **42** (1993), 203–210.
15. W. Hohberg, How to find biconnected components in distributed networks, *J. Parallel Distrib. Comput.* **9** (1990), 374–386.
16. S. T. Huang, A new distributed algorithm for the biconnectivity problem, *in* "Proceedings of the 1989 International Conference on Parallel Processing, Vol. III, Algorithms and Applications," pp. III-106–III-113. University Park, Penn, Penn State, August 1989.
17. S. T. Huang, Leäder election in uniform rings, *ACM Trans. Program Languages Systems* **15** (1993), 563–573.
18. S. T. Huang and N. S. Chen, A self-stabilizing algorithm for constructing breadth-first trees, *Inform. Process. Lett.* **41** (1992), 109–117.
19. M. H. Karaata, A self-stabilizing algorithm for finding articulation points, *Int. J. Foundations Comput. Sci.* **1** (1999), 33–46.
20. M. H. Karaata and P. Chaudhuri, A self-stabilizing algorithm far bridge finding, *Distrib. Comput.* **2** (1999), 47–53.
21. S. Katz and K. Perry, Self-stabilizing extensions for message-passing systems, *Distrib. Comput.* **7** (1993), 17–26.
22. X. Lin and S. Ghosh, Self-stabilizing maxima finding, *in* "Proceedings of the 28th Annual Allerton Conference," pp. 662–671, 1991.
23. M. Schneider, Self-stabilization, *ACM Comput. Surv.* **25** (1993), 45–67.
24. B. Swaminathan and K. J. Goldman, An incremental distributed algorithm for computing biconnected components in dynamic graphs, *Algorithmica* **22** (1998), 305–329.
25. J. Woo and S. Sahni, Computing biconnected components on a hypercube, *J. Supercomput.* **5** (1991), 73–87.

MEHMET HAKAN KARAATA received his Ph.D. in computer science in 1995 from the University of Iowa. He joined Bilkent University, Ankara, Turkey, as an assistant professor in 1995. He is currently working as an associate professor in the Department of Computer Engineering, Kuwait University. His research interests include mobile computing, distributed systems, fault-tolerant computing and self-stabilization.